NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

ANALYSIS OF INTEL IA-64 PROCESSOR SUPPORT FOR A SECURE VIRTUAL MACHINE MONITOR

by

Kadir Karadeniz

March 2001

Thesis Advisor:	Cynthia Irvine
Second Reader:	Frederick W. Terman

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704-0188*
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** March 2001	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE: Analysis of Intel IA-64 Processor Support for a Secure Virtual Machine Monitor			**5. FUNDING NUMBERS**
6. AUTHOR(S) Karadeniz, Kadir			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.			**12b. DISTRIBUTION CODE**

13. ABSTRACT ***(maximum 200 words)***

This thesis explores the Intel IA-64 architecture's capability to support a secure virtual machine monitor. The major mission of a virtual machine monitor is to provide an execution environment identical to the real machine environment for virtual machines. A VMM duplicates the real resources of a processor for virtual machines while making a virtual machine think that it is running on a real machine. As a result, a virtual machine monitor allows multiple virtual machines to run concurrently on the same machine.

A secure VMM on the Intel IA-64 architecture would offer several benefits. A secure VMM would ensure that security policy is enforced by constraining information flow between the supported virtual machines. This would provide PC users with a more secure environment in which to run COTS operating systems.

The Intel IA-64 architecture was analyzed to determine if it is virtualizable. Three types of virtual machine monitors and their hardware requirements have been defined. The IA-64 architecture was mapped to these hardware requirements. Analysis showed that the IA-64 architecture meets three main hardware requirements. However, IA-64 instruction set contains 18 sensitive unprivileged instructions. These instructions prevent the IA-64 architecture from being used for a Type I VMM. Several virtualization techniques used in some architectures are discussed to determine if these techniques could be applicable to virtualization of the IA-64 architecture.

14. SUBJECT TERMS Virtual Machines, Virtual Machine Monitors, Intel IA-64 Architecture			**15. NUMBER OF PAGES** 118
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

ANALYSIS OF INTEL IA-64 PROCESSOR SUPPORT FOR A SECURE VIRTUAL MACHINE MONITOR

Kadir Karadeniz
Lieutenant J. G., Turkish Navy
B.S., Turkish Naval Academy, 1995

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
March 2001

Author: [illegible]
Kadir Karadeniz

Approved by: Cynthia E. Irvine
Cynthia Irvine, Thesis Advisor

Frederick W. Terman
Frederick W. Terman, Second Reader

[illegible]
Jeffrey B. Knorr, Chairman
Department of Electrical and Computer Engineering

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

This thesis explores the Intel IA-64 architecture's capability to support a secure virtual machine monitor. The major mission of a virtual machine monitor is to provide an execution environment identical to the real machine environment for virtual machines. A VMM duplicates the real resources of a processor for virtual machines while making a virtual machine think that it is running on a real machine. As a result, a virtual machine monitor allows multiple virtual machines to run concurrently on the same machine.

A secure VMM on the Intel IA-64 architecture would offer several benefits. A secure VMM would ensure that security policy is enforced by constraining information flow between the supported virtual machines. This would provide PC users with a more secure environment in which to run COTS operating systems.

The Intel IA-64 architecture was analyzed to determine if it is virtualizable. Three types of virtual machine monitors and their hardware requirements have been defined. The IA-64 architecture was mapped to these hardware requirements. Analysis showed that the IA-64 architecture meets three main hardware requirements. However, IA-64 instruction set contains 18 sensitive unprivileged instructions. These instructions prevent the IA-64 architecture from being used for a Type I VMM. Several virtualization techniques used in some architectures are discussed to determine if these techniques could be applicable to virtualization of the IA-64 architecture.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS/DEDICATIONS

One of the great pleasures of finishing up this thesis is acknowledging the support of many people whose names may not appear anywhere in the thesis, but whose cooperation, friendship, understanding and patience were crucial for me to prepare this thesis and successfully publish it.

I would like to thank my thesis advisor Prof. Cynthia Irvine and my second reader Senior Lecturer Frederick W. Terman for their patience. Additionally, I would like to thank my friends Seval and Bugra. Without their support and encouragement, I would not be able to create a thesis like this.

Finally, I would like to dedicate this thesis to my parents Alisait, Saniye, brother Atilla, sister-in-law Ayla and niece Yagmur for their sincere love and confidence.

THIS PAGE INTENTIONALLY LEFT BLANK

EXECUTIVE SUMMARY

A significant revolution in computer technology has taken place over the past 30 years. The advent of the modern operating systems shielded programmers and typical users from the complexity of the hardware. When users or their programs require services to be performed, they call the operating system which performs the actual services and resource allocation. However, system programmers desire or need direct access to the real resources of the system. Their programs cannot run under the control of the normal operating system. Therefore, to do their jobs, system programmers have to both work in the computing facility and make the system inaccessible to other users. Virtual machine monitors were invented to resolve these difficulties.

Basically, a virtual machine (VM) is an identical and efficient copy of the real machine. Each virtual machine can be considered a separate operating system platform. A virtual machine monitor (VMM) mediates between the virtual machines and the real machine, and provides an execution environment identical to the real machine environment for the virtual machines. Making a virtual machine think that it is running on a real machine has many benefits. First, a user can run multiple operating systems on the same computer concurrently. Second, a user can test the new versions of an operating system without having to completely reboot the entire system from the beginning. Third, virtual machines can be used to upgrade an operating system without corrupting the original operating system. Fourth, a user can run applications on different environments without having to use another real machine. Finally, virtual machine monitors provide an environment for interactive debugging of an operating system.

The goal of this thesis is to determine whether any type of VMM can be built on the IA-64 architecture. If a VMM can be built on the IA-64 architecture, its security will be examined.

Three types of virtual machine monitors are discussed. A Type I VMM runs on a real machine and can be treated as an operating system that is responsible for providing an execution environment and resource allocation for VMs. A Type II VMM runs as an application under a host operating system and provides only virtualization services. The host operating system handles the resource allocation and provides the extended environment for the VMM. A Hybrid Virtual Machine (HVM) is identical to a VMM. The major differences between an HVM and a VMM are that, in a HVM, all privileged instructions are interpreted by software while in a VMM, privileged instructions can execute directly on the real processor.

In order to support a VMM, a processor must be able to be virtualized. If all sensitive instructions, which cause interpretation problems and interfere with underlying VMM when they are executed directly by a VM, are also privileged, then the processor is said to be virtualizable. This means all sensitive, privileged instructions will trap to the VMM.

The hardware and software requirements for virtualization are given for each type of VMM. Then, the Intel IA-64 architecture is analyzed to determine if it is virtualizable. The Intel IA-64 architecture was designed to overcome the performance limitations of today's architectures. With some innovative features such as speculation, predication and register rotation, the IA-64 architecture can address the requirements of high-end workstation and server applications. The IA-64 architecture was mapped to hardware requirements. Analysis showed that, the IA-64 architecture meets three main hardware requirements to support a Type I VMM. That is;

(1) There is no difference between the methods of executing instructions in supervisor and user modes in the IA-64 architecture.

(2) The IA-64 has mechanisms, such as a protection system and address translation system that protects the real and virtual machines from the active VM.

(3) The IA-64 architecture uses interrupts, faults and traps that cause the flow of control to be passed to an interrupt handling routine when an unprivileged task executes a privileged instruction.

However, the IA-64 instruction set contains sensitive, unprivileged instructions. Since the unprivileged instructions do not generate an interrupt, fault or trap, the VMM will have no opportunity to simulate the behavior of the instruction.

Each of the IA-64 architecture's instructions documented as of July 2000 was examined. The examination found 18 instructions that make the IA-64 architecture non-virtualizable. The table below lists the 18 instructions and the reason why they do not meet VMM requirements.

INSTRUCTIONS	REASON FOR VMM REQUIREMENTS VIOLATION							
	3a1	3a2	3b1	3b2	3c1	3c2	3c3	3d
br.call, br.ret, cmpxchg, probe, fetchadd, xchg, ld, ldf, ldfp, st, stf					X			
fc, lfetch					X	X		
cover, mov um, rum, sum			X					
br.ia	X							

The reasons for violation of the VMM requirements are as follows:

3a1 : References or changes the mode of the processor.

3a2 : References or changes the state of the processor.

3b1 : References or changes sensitive registers.

3b2 : References or changes sensitive memory locations.

3c1 : References or changes privilege level.

3c2 : References the memory system.

3c3 : Interferes with address translations.

3d : I/O instructions.

Several architectures including the VAX security kernel, Intel 80x86, DISCO and Alpha, and virtualization techniques used for virtualization of these architectures are discussed to determine if these techniques would also be applicable to virtualizing the IA-64 architecture.

After current VMM architectures and virtualization techniques were analyzed, it became clear that some slight modifications to the Intel IA-64 architecture could make virtualization easier. First, all 18 sensitive, unprivileged instructions could be redesignated as privileged instructions. A second modification could be to implement a trap on op-code instruction.

A secure VMM on the IA-64 architecture would offer several benefits. First, it could eliminate the need for dedicated systems for each security level. Second, PC users would be able to work in a high assurance system with commercial-off-the-shelf (COTS) operating systems.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. BACKGROUND

A modern computer system is a complex system. It consists of one or more processors, main memory, terminals, network interfaces and input/output devices. Writing programs that manage and use these components is extremely difficult. Many years ago, it became clear that a solution had to be found to shield programmers and typical users from the complexity of the hardware. The way that has evolved is to put a layer of software on top of the real hardware to manage all components of the system and provide users with an interface that is easier to understand. This solution, known as an operating system, eliminated the need for direct access to the real system resources by a user's program.

However, system programmers need direct access to the real resources of the system. Unlike a user's programs, their programs have to be run on a real machine and cannot run under the control of the normal operating system. Therefore, to do their jobs, system programmers have to both work in the computing facility and make the system inaccessible to other users. Virtual machine monitors (VMMs) were invented to resolve these difficulties.

A virtual machine monitor is software that mediates between the virtual machine and the real machine. A virtual machine monitor allows multiple virtual machines to run concurrently on a single computer by managing the real resources and allocating these real resources to the virtual machines as needed. Basically, a virtual machine (VM) is an efficient and identical copy of the real machine. Each virtual machine can be considered a separate operating system platform. A secure VMM would ensure that security policy is enforced by constraining information flow between the supported virtual machines (VMs). Two operating systems could be at different security levels and sharing between the virtual machines would be possible only as allowed by security policy.

Intel designed a processor called Itanium^{TM} (formerly code-named Merced), based on the IA-64 architecture to overcome the performance limitations of today's architectures. With some innovative features such as speculation, predication, branch architecture and register rotation, the IA-64 architecture could address the requirements of high-end workstation and server applications. If a secure VMM could be built for the IA-64 architecture, it would be very powerful. It would be useful to understand the feasibility of building a secure VMM on the Intel IA-64. The ability of the processor to support a secure VMM could determine how future secure network architectures might be designed.

A secure VMM on the Intel IA-64 architecture would offer several benefits. First, it could eliminate the need for separate systems for each classification level. Second, computer users would be able to work in a highly secure environment and still run commercial-off-the-shelf (COTS) software applications. Intel has not stated if a VMM can be built on the Intel IA-64 architecture.

B. THESIS GOAL

The goal of this thesis is to determine whether any type of VMM can be built on the IA-64 architecture. If a VMM can be built on the IA-64 architecture, its security will be examined.

This work requires a detailed understanding of the processor's organization and architecture as well as its instruction set and also involves an understanding of virtual machine monitors.

C. RELATED WORK

There are several works related to this thesis. Most of the information about the virtual machine monitor concept is given by Goldberg [Ref. 7]. In his thesis, Goldberg gives the definition of the virtual computer systems. He introduces three types of virtual machine monitors and develops hardware and software requirements for virtualization in order to implement a virtual machine monitor on a processor's hardware.

Also, Popek introduces formal requirements for virtualizable third generation architectures [Ref. 8]. He describes the essential characteristics and modules of the virtual machine monitors.

Several previous efforts constructed VMMs. The VAX security kernel is one of them. The VAX security kernel was a research effort that resulted in a VMM for the VAX architecture [Ref. 12]. The VAX security kernel creates virtual VAX processors and provides isolation and controlled sharing of sensitive information. The VAX security kernel applies mandatory and discretionary access controls to virtual machines.

Other work analyzes the Intel architecture. Lawton proposes several virtualization techniques that allow multiple operating systems to run concurrently on an IA32 PC [Ref. 21]. He developed a product called FreeMWare (renamed plex86) that provides something similar to a VMM for the Intel architecture. On the other hand, in his thesis, Robin analyzes the Intel Pentium's capability to support a secure virtual machine monitor [Ref. 11]. He states that the Intel architecture is not virtualizable. Robin concludes that even though the FreeMWare provides something similar to a VMM for the Intel architecture, none of the VMMs for the Intel architecture should be considered a secure VMM.

Today, there is no related work that discusses Intel IA-64 architecture support for a secure VMM. This thesis builds upon previous and related work that discusses the requirements for virtualization, and virtualization techniques to contribute to our understanding of the Intel IA-64 architecture's capability to support a secure VMM.

D. THESIS ORGANIZATION

The remainder of this thesis is divided into five chapters. Chapter II contains a brief summary of the Intel IA-64 architecture. The new features, application and system state, addressing and protection mechanisms of the IA-64 architecture are addressed in this chapter. In Chapter III, VM and VMM concepts are introduced. Three different types of VMMs and their hardware virtualization requirements are given. Chapter IV examines all of the instructions of the IA-64 instruction set in order to determine if the IA-64 architecture is virtualizable. The instructions that violate the hardware virtualization requirements are discussed in more detail. Chapter V analyzes whether or not the IA-64 architecture can be made virtualizable. Several architectures such as the VAX security kernel, the Intel x86, the Disco and the Alpha are discussed in order to present several virtualization techniques. Finally, chapter VI provides a conclusion for this thesis work and also addresses future work.

II. INTEL IA-64 ARCHITECTURE

As technology rapidly expands, so does the need for higher performance computing. Today, computer architecture must develop at a pace that will enable hardware to meet the growing demands of increasingly sophisticated applications and operating systems. Applications such as Internet communication, e-commerce and data warehousing make heavy demands on high performance workstations and servers. While the performance of today's processors continues to improve, architectures based on out-of-order execution models, that is, processors that use multiple execution units to execute instructions in non-sequential order (i.e., instruction 2 can execute before instruction 1 has completed), require complex hardware mechanisms to make sure that results of execution are reassembled into the correct order to ensure that program runs without error. Inefficient use of these models may result in undesirable effects that can severely limit performance, such as memory latency and mispredicted branches.

To overcome the performance limitations of existing architectures, the Intel-designed IA-64 architecture employs innovative techniques and features. These techniques and features, including speculation, predication, large register files, register rotation, advanced branch architecture and 64-bit memory addressability, are combined to address the modern requirements of high end workstation and server applications.

In order to analyze Intel IA-64 processor support for a secure virtual machine monitor, this chapter will provide a brief overview of Intel IA-64 architecture and its innovations. For more information about Intel IA-64 architecture, see the Intel's manuals [Ref. 1,2,3,4].

A. INNOVATIONS OF THE IA-64 ARCHITECTURE

The IA-64 architecture has several innovative features such as speculation, predication, branch architecture, register rotation and register stack. These features will be described below.

1. Speculation

Memory latency is one of the performance limitations in standard architectures. In fact, memory speed is much slower than processor speed, and yet the processor must load data from memory as early as possible to guarantee that the data is available when needed. The IA-64 architecture allows a compiler to initiate loads much earlier, even before it is known that the information contained in these loads will be needed. This process is called *speculation*. The IA-64 architecture provides two types of speculation, data speculation and control speculation.

a. Data Speculation

Data speculation is the execution of a load instruction prior to a store that might write to the same memory address from which the load reads. Early scheduling of loads are critical because loads bring values from memory to registers. Memory values may be needed to start a computation in a program, therefore, loading memory values to registers must be performed as early as possible. Data speculative loads are also called advanced loads.

To detect the address collisions between advanced loads and stores, a structure called Advanced Load Address Table (ALAT) is used by operating system. The code in Figure 1 below shows the data speculation of a load. When an advanced load *(aload)* is executed, an entry which contains the register tag, a portion of the physical memory address being loaded and the load access size in bytes, is inserted into the ALAT. When a store instruction is executed, the ALAT is searched in one of the two different ways: by physical address or by ALAT register tag, which is a unique number obtained from the physical target register number. If an entry with a matching physical address or register tag is found, it is deleted from the ALAT.

When compiler executes with data speculation, it places a check instruction *(acheck)* at the original location of the load instruction. The check instruction searches the ALAT for the specified register tag or physical address. If ALAT contains an entry, then the data speculation was successful. If there is no ALAT entry, then the check instruction branches to recovery code that reloads the value and writes it into the target register. It is the compiler's responsibility to generate the recovery code if the speculative load fails.

Normal code sequence;	Data speculative code sequence;
store (store address, data)	*aload (load address, target)*
load (load address, target)	*/* operations that use target */*
use (target)	*store (store address, data)*
	acheck (target, recovery address)
	use (target)

Figure 1. Data Speculation After Ref. [1].

b. Control Speculation

Control speculation is the execution of loads before the guarding branch instruction to hide memory latency. When the compiler encounters a branch instruction, the load is moved before the branch condition statement. This can cause undesired effects such as exceptions if the memory address is not valid. If memory access fails to occur when a speculative load executes, then a bit termed, "*deferral token*" is set in the target register. A deferral token for general registers is the 65th bit that is called "NaT" (Not a Thing) bit. A floating-point register uses special encoding for its deferral token called, "NatVal" (Nat a Thing Value). Arithmetic operations propagate the deferral token. As an example, a subtract operation of two general registers also performs a logical "or" operation on the NaT bit. If either one of the NaT bits of these registers is set, then the subtract operation sets the NaT bit of the destination register. Since, just the general registers and the floating-point registers have deferral tokens, destination registers other than general and floating-point registers cannot propagate NaT bits. The compiler therefore must not move the instructions to speculative locations if the destination is not a general or floating-point register.

The code in Figure 2 below clarifies the control speculation process. When a compiler executes control speculation *(sload)*, it puts a check instruction *(scheck)* at the original location of the load instruction. If the NaT bit of the target register is clear, then the control speculation was successful. If the NaT bit is set, then the check instruction branches to a recovery code.

```
Normal code sequence;               Control speculative code sequence;
if (x==y)                           sload (load address1, target1)
load (load address1, target1)       sload (load address2, target2)
else                                /* operations that use target1 and target2*/
load (load address2, target2)       if (x==y)
                                    scheck (target1, recovery address1)
                                    else
                                    scheck (target2, recovery address2)
```

Figure 2. Control Speculation After Ref. [1].

2. Predication

Another performance limitation for existing architectures is mispredicted branches. At run time, conditional branches are difficult to execute, because a processor must decide what instructions to fetch beyond a branch before it computes the condition for that branch. Predication is the conditional execution of the instructions, and removes branches and associated mispredicts.

The code in Figure 3 below summarizes the predication process. The IA-64 executes the conditional branch instructions (e.g., compare x==y) leading to a branch and puts the result, whether the comparision is true or false, in two one-bit predicate registers (e.g., p1 and p2). These two predicate registers are the complement of each other. The execution of the branch depends on the predicate registers. If the predicate register is set, that is the result of comparision is true (p1=1, p2=0), then the instruction executes normally. If the predicate register is clear, that is the result of comparision is false (p1=0, p2=1), then the instruction does not execute, and control passes to the next instruction.

```
Normal code sequence;          Predicated code sequence;

if (x==y)                      p1, p2 = compare (x==y)

x=x+y                          (p1) x=x+y

else                           (p2) x=x-y

x=x-y
```

Figure 3. Predication After Ref. [1].

3. Branch Architecture

Efficient execution of the branches is another critical performance issue. When a branch instruction is encountered, the processor fetches the next instruction from another address rather than the next sequential address. The IA-64 architecture defines two types of branches, indirect branches and instruction pointer (IP) relative branches. For IP relative branches, the IP is incremented by a signed 21-bit value (offset), provided from the encoding of the branch instruction opcode. For indirect branches, a value is assigned to the IP from one of the eight 64-bit branch registers, which is provided from the encoding of the branch instruction opcode.

For optimization of branches, the IA-64 defines multi-way branches. If an instruction group has instructions after the branch instruction, then these instructions will execute only if the branch is not taken. In this way, a compiler can schedule compares and dependent branches in the same pipeline cycle.

Traditional branches always mispredict the last iteration of a loop. If a loop executes five iterations, and the branch prediction mechanism mispredicts the last iteration, as a result, the prediction rate for this loop will be reduced, and this will reduce the effective speed of the processor. The IA-64 also provides special mechanisms for loop closing branches for optimization purposes. A loop closing branch is implemented with the *cloop* instruction. This instruction checks the Loop Count (LC) register which contains the number of iterations the loop will execute. If the register value is not zero, then the value is decremented by one and control passes back to the beginning of the loop. If the value is zero, then no action is taken, and control passes to the next instruction. This operation makes loop termination perfectly predictable.

4. Register Rotation

Efficient handling of the loops is one of the important determiners of the processor's performance. The IA-64 provides register rotation for general registers (GRs), floating-point registers (FRs) and predicate registers (PRs) to be able to execute loop iterations in parallel rather than sequentially. The table below shows the static and rotating subsets of these registers.

Register Name	Static Registers	Rotating Registers
General Registers	GR-0 through GR-31	GR-32 through GR-127
Floating point Registers	FR-0 through FR-31	FR-32 through FR-127
Predicate Registers	PR-0 through PR-15	PR-16 through PR-63

Table 1. Static and Rotating Registers

Register rotation allows loop optimization by scheduling multiple copies of the loop body together. A special register holds the register rotation base (RRB) which is used to rename the registers within the rotating subset of each register set. A reference to any register within the range of rotating subset is offset by the value of the RRB. For example, if the RRB has a value of 5, a reference to GR-32 would actually refer to GR-

37. The RRB value wraps the register values through the use of modulo arithmetic. Therefore the register values appear to rotate.

Register rotation is the base of modulo-scheduling (i.e., new iterations can begin before previous iterations have finished.) Modulo-scheduling allows multiple loop iterations to be overlapped, and thus, increases parallelism. In a simple loop execution, the throughput is 2 cycles/word, and three instructions can execute in those two cycles. On the other hand, in a modulo-scheduled loop execution, the throughput is 1 cycle/word, and three instructions can execute each cycle.

Cycle	Loads	Stores	Loop Closing Branches
1	ld1		
2	ld2	st1	br.ctop1
3	ld3	st2	br.ctop2
4	ld4	st3	br.ctop3
5	ld5	st4	br.ctop4
6		st5	br.ctop5

Figure 4. Modulo-Scheduled Loop Execution Sequence From Ref. [5].

A Modulo-scheduled loop contains three code sections: Prologue, Kernel and Epilogue. The prologue code fills the software pipeline. The software pipeline is analogous to hardware pipelining, but is created in code by the compiler. The Kernel code starts and finishes one iteration. The Kernel code is repeated for iteration counts greater than two. The Epilogue code terminates the loop iterations, and drains the software pipeline.

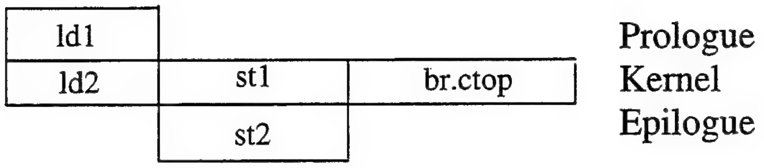

Figure 5. Schedule of Modulo-Scheduled Loop From Ref. [5].

For counted loops, the IA-64 uses special loop branches (e.g., *br.ctop* instruction) and two application registers to determine the number of iterations required to drain the software pipeline. If LC>0, then LC is decremented by one and a branch is taken. When LC==0, the Epilog Count (EC) Register is tested. If EC>1, then EC is decremented by one and the branch is taken. When both LC and EC==0, the branch falls through.

Combining register rotation and predication increases loop optimization. Figure 6 shows the predicated modulo-scheduled loop execution sequence for n iterations. In the execution sequence, one predicate register (e.g., p16) is initialized to one, the other predicate register (e.g., p17) is initialized to zero. In the first cycle, the load executes but the store does not. In second cycle, after the first *br.ctop* executes, the predicates have been rotated so that the p16 value is now in p17. Also, *br.ctop* wrote one into p16. This continues until cycle n is reached. In cycle n, LC has zero value and br.ctop still branches back because EC is not zero yet. The *br.ctop* writes zero into p16 and p17 still has a one, this combination has the effect of shutting off the load and allowing the final store.

The IA-64 also provides optimizations for efficient modulo-scheduling of *while* loops. The IA-64 has a *br.wtop* instruction that executes a loop as long as a given predicate is one. When the branch predicate is zero, the branch continues to execute the loop while decrementing EC until EC==0.

	Stage Predicate			Predicate Value			
Cycle	P16	P17		P16	P17	LC	EC
1	ld1		br.ctop	1	0	n-1	2
2	ld2	st1	br.ctop	1	1	n-2	2
3	ld3	st2	br.ctop	1	1	n-3	2
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
n	ldn	stn-1	br.ctop	1	1	0	2
n+1		stn	br.ctop	0	1	0	1
				0	0	0	0

Figure 6. Predicated Modulo-Scheduled Loop Execution Sequence From Ref. [5].

5. Register Stack

In the IA-64 architecture, general registers are divided into two subsets, a static subset and a stacked subset. Registers GR-0 through GR-31 are called static registers. The stacked registers are numbered from GR-32 through GR-127. The static subset must be saved and restored on context switches. The stacked subset is automatically saved and restored by the Register Stack Engine (RSE) which moves registers between the register stack and a storage location in memory.

As a side effect of procedure calls and returns, the register stack is implemented by renaming the registers, that is, register access is accomplished by renaming the virtual register identifiers in the instruction through a base register into the physical register. By renaming the registers, there will be no need to spill (i.e., to write the register contents to memory), and fill (i.e., to write the memory contents back to registers), on subroutine calls and returns. The callee can use available registers without spilling and restoring the caller's registers.

The registers in a stacked subset are called a register stack frame. The frame (sof) is divided into two parts: the local area (sol) and the output area (sof-sol). A callee procedure defines the size of its new stack frame using the *alloc* instruction. By using this instruction, the callee can allocate up to 96 registers per frame for input, output and local values.

Figure 7 describes the behavior of the register stack on a procedure call from procedure A to procedure B. The behavior of the register stack is shown at four points: prior to the call, immediately after the call, after callee (procB) executed an *alloc* instruction, and after return. After a call, the Current Frame Marker (CFM) is copied to the Previous Frame Marker (PFM) field in the Previous Function State application register. The new frame's local area size is set to zero, and the size of the callee's output area (output B1) is set to the size of the caller's output area (output A), thus allowing parameters to be passed with no register copying.

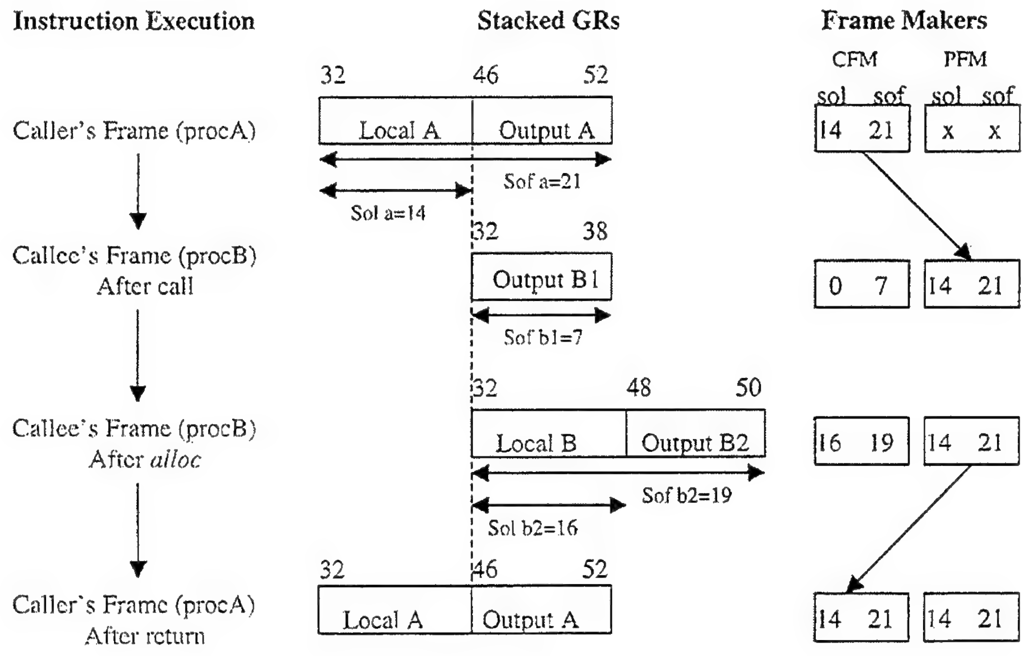

Figure 7. Register Stack Behavior During Procedure Call and Return From Ref. [1].

6. Other Innovations

IA-64 architecture has several other innovations and features, including:

- Support for binary compatibility with the IA-32 instruction set.
- Support for Instruction Level Parallelism that has the ability to execute multiple instructions at the same time.
- Mechanisms for the compiler to communicate with the processor about compile time information.
- Multimedia instructions.

B. APPLICATION REGISTER STATE

IA-64 contains a rich set of registers available to application programs. The names of these registers and their functions are described below.

1. General Registers (GRs)

IA-64 contains a set of 128 general registers, each of them has 64 bits of data storage and one NaT bit. General registers are used for integer and integer multimedia operations. When executing an IA-32 instruction set, general registers 8 through 31 contain the IA-32 integer, segment descriptor and segment selector registers.

2. Floating-Point Registers (FRs)

IA-64 contains a set of 128 floating-point registers, each of them has 82 bits. These registers are used for all floating-point operations.

3. Predicate Registers (PRs)

IA-64 has a set of 64 predicate registers, each of them has one bit. Predicate registers are used for conditional execution of instructions. The results of IA-64 compare instructions are held in these registers.

4. Branch Registers (BRs)

There are 8 branch registers in IA-64 architecture, each of them has 64 bits. These registers contain IA-64 branching information and provide branch target addresses for indirect branches.

5. Instruction Pointer (IP)

In the IA-64 architecture, three instructions are grouped together into 128-bit containers called bundles. The instruction pointer (IP) contains the address of the bundle that holds the currently executing instruction. The IA-64 instruction bundles are 16 bytes, and are 16-byte aligned. Therefore, the least significant 4 bits of IP are always zero.

6. Current Frame Marker (CFM)

The Current Frame Marker contains the state of current stack frame. On a subroutine call, the CFM is copied to the Previous Frame Maker (PFM) field in the Previous Function State Register which holds information about previous state. The PFM is saved and restored for every procedure in case it gets modified, such as by subsequent subroutine calls. This provides each general register stack frame with a different frame marker.

7. Performance Monitor Data Register

This register is used to obtain performance information for execution of both IA-64 and IA-32 instructions.

8. User Mask (UM)

This register is used to control memory access alignment and byte ordering that defines whether loads and stores use little-endian or big-endian byte ordering. It is also used to enable or disable the user-configured performance monitor used to analyze both hardware and software performance. Lastly, it holds IA-64 floating-point register state modifications.

9. Processor Identification Registers

These registers are also called CPUID. CPUID registers 0 and 1 contain vendor information. CPUID register 2 is an ignored register. CPUID register 3 contains version information and CPUID register 4 provides general application level information about IA-64 features. It is a set of flag bits that shows if a given IA-64 is supported in the processor model.

10. Application Registers

The IA-64 provides a rich set of special purpose data and control registers for processor functions visible to applications.

Figure 8 shows the IA-64 application register file.

C. SYSTEM REGISTER STATE

The system registers of the IA-64 architecture are privileged system registers used for debugging, process control, protection, handling interrupts, and performance monitoring. The names of these registers and their functions are described below.

1. Processor Status Register (PSR)

This is 64-bit register that holds the control information for the currently executing IA-64 or IA-32 process.

2. Control Registers (CRs)

The IA-64 has several control registers that hold the current state of process and global processor parameters for interrupt handling and memory management. Figure 9. shows all the control registers.

3. Debug Breakpoint Registers (DBR/IBR)

An IA-64 operating system can program 64-bit data and 64-bit instruction breakpoint register pairs to invoke faults on references to physical and virtual addresses by IA-64 and IA-32 instructions.

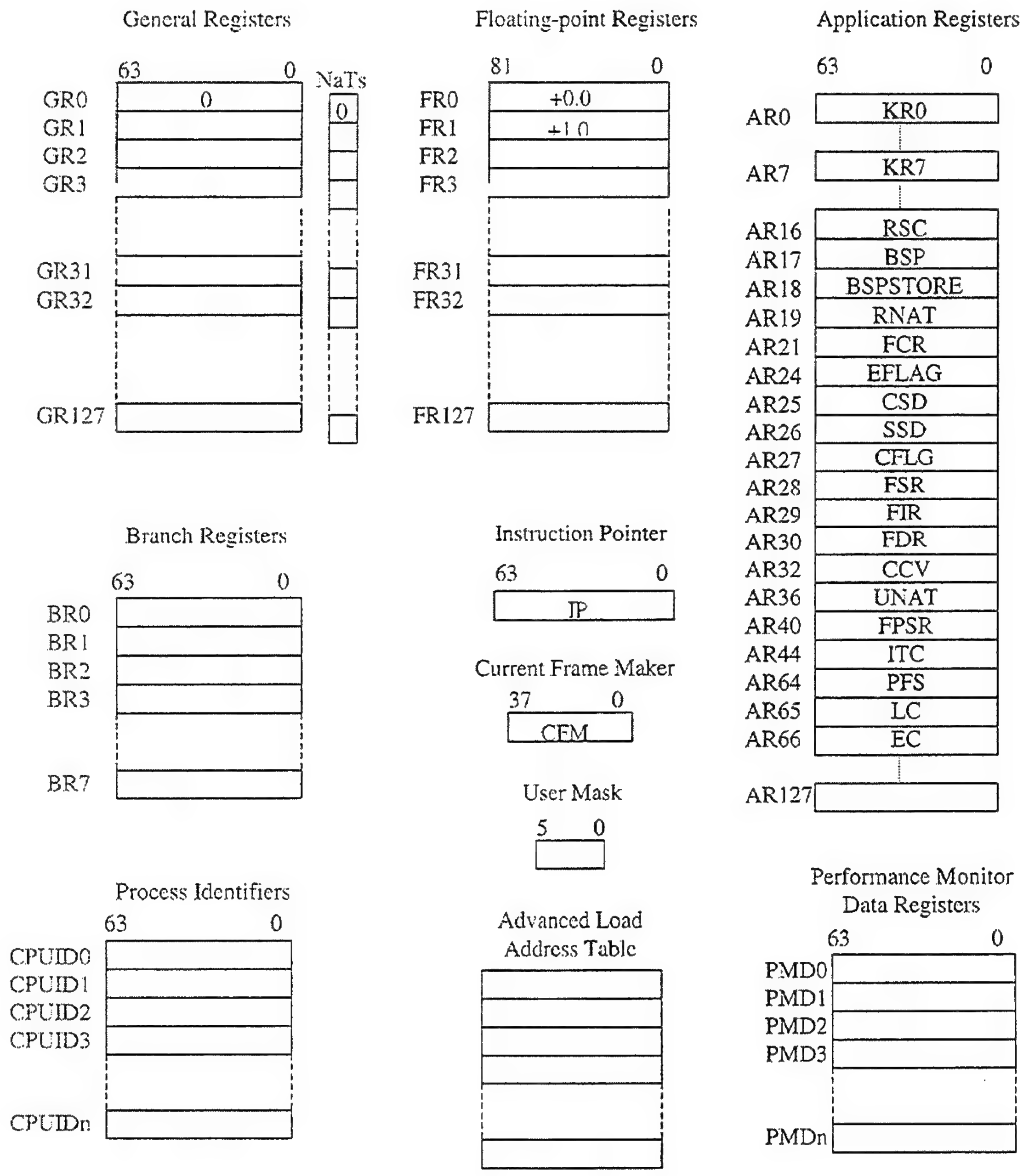

Figure 8. Application Register Set From Ref.[1].

4. Performance Monitor Configuration/Data Registers (PMC/PMD)

These registers can be programmed to measure application, operating system and processor performance.

5. Banked General Registers

IA-64 uses GR-16 through GR-31 as banked general registers. Banked general registers provide immediate register context for low level interrupt handlers (i.e., TLB miss handler). Upon interrupt, the processor switches general registers GR-16 through GR-31 to register bank 0.

6. Region Registers (RRs)

Eight 64-bit region registers hold the region identifier (RID) values and preferred page sizes for virtual to physical address mappings.

7. Protection Key Registers (PKRs)

Sixteen 64-bit protection key registers hold the protection keys and read, write and execute permissions for protection domains. These registers provide a register cache for the protection keys required by a process. The operating system is responsible for management and replacement policies of these registers.

8. Translation Lookaside Buffer (TLB)

The translation lookaside buffer holds the virtual and physical address mappings that are currently in use.

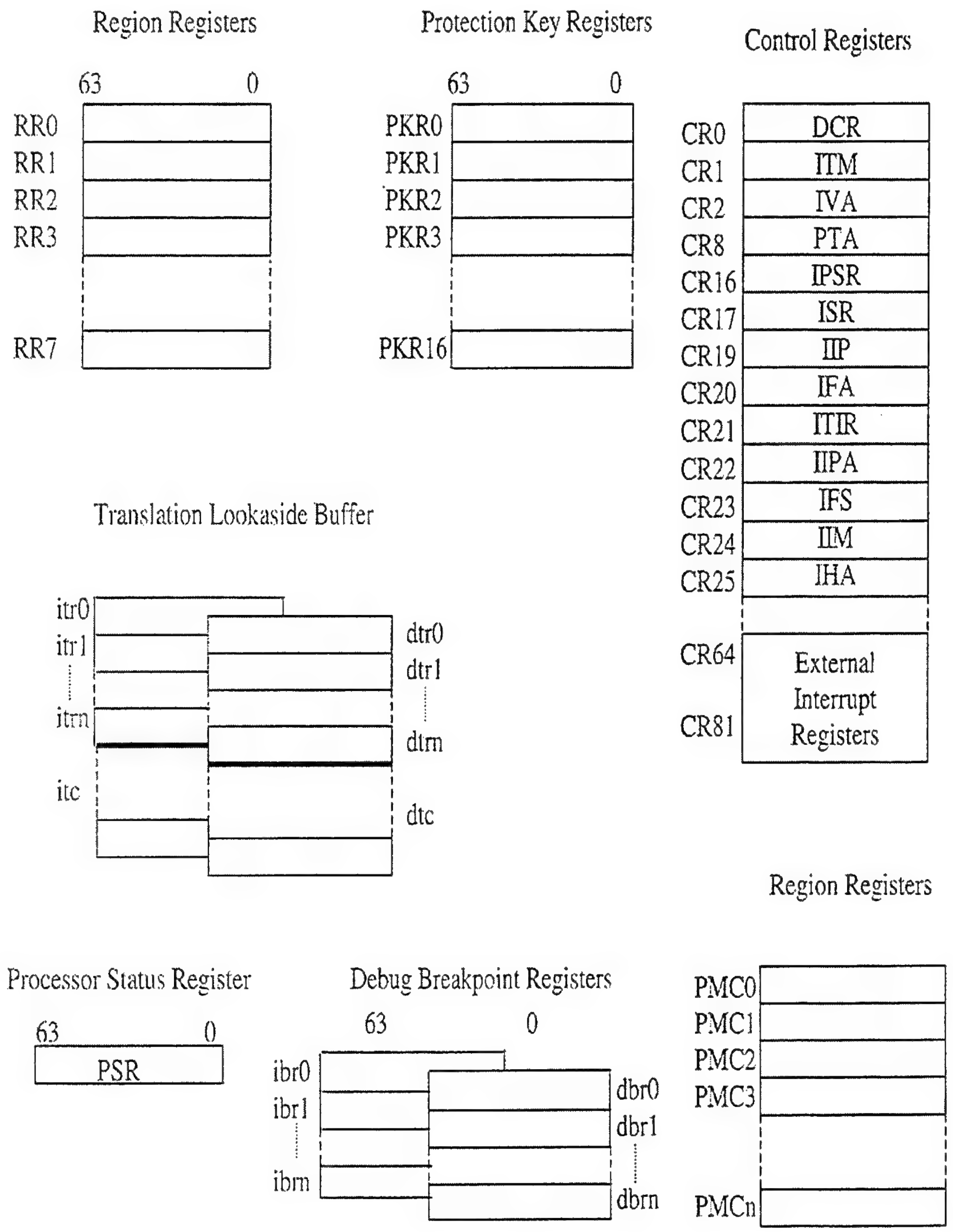

Figure 9. System Register Set From Ref. [2]

D. ADDRESSING AND PROTECTION

1. Memory Models and Virtual Addressing

Operating Systems are generally built on one of two virtual memory models, the multiple address space model (MAS) or the single address space (SAS) model.

In MAS models, each process has a unique address space, whereas in SAS models, more than one process can exist within the same virtual address space.

The IA-64 architecture defines virtual regions to support both the MAS and SAS models. Each processor implements 2^{24} virtual regions, each 2^{61} bytes in size. The three most significant bits of the virtual address selects one of eight regions. The operating system maps these eight regions onto the system's 2^{24} possible regions through region identifiers which are loaded into eight region registers before the process executes. These registers are saved and restored during context switches. Region identifiers can be thought of as address space identifiers for MAS systems, or as the most significant address bits of 2^{85} bytes of global virtual address space for SAS systems.

Figure 10 shows the process of mapping a virtual address onto a physical address. On a memory reference, a virtual region number (VRN) selects one of the eight region registers which contain RIDs. The virtual page number (VPN) and RID work together to search the TLB for a translation entry. If a match is found in the translation lookaside buffer (TLB), then the translation entry in the TLB is used. The entry's physical page number (PPN) is concatenated to page offset bits to form the physical address. If a matching entry is not found, the processor may optionally search the virtual hashed page table (VHPT).

VHPT exists in two forms. The first, region-based VHPT short format, uses an 8-byte entry format and is directly indexed by the VPN. This format is generally used to support a per region linear page table configuration. The second, long format VHPT, uses 32-byte entry format and is indexed by a hash function. This format is suitable for a single large virtual hash page table. Figure 11 shows the sequence of the TLB and VHPT search.

2. Protection Mechanisms

The IA-64 architecture provides four types of protection mechanisms [Ref. 6]. These mechanisms are addressability, privilege level, access rights and protection domains. Each will be discussed below.

Addressability is a protection mechanism such that one process cannot access the memory address of another process. As mentioned before, each process is assigned a RID which points to a memory space for that process. Manipulation of RIDs by trusted software restricts one process from addressing the memory space of another process. The operating system can grant shared access to a particular region by assigning the same RID to the processes. It is the responsibility of the operating system to use protection keys to manage multiple accesses to a shared page with different access rights.

Four privilege levels are defined in the IA-64 architecture, numbered from zero to three, to control access to system resources. Level zero is the most privileged, while level three is the least privileged. Level zero is the only level from which the privileged instructions can be executed.

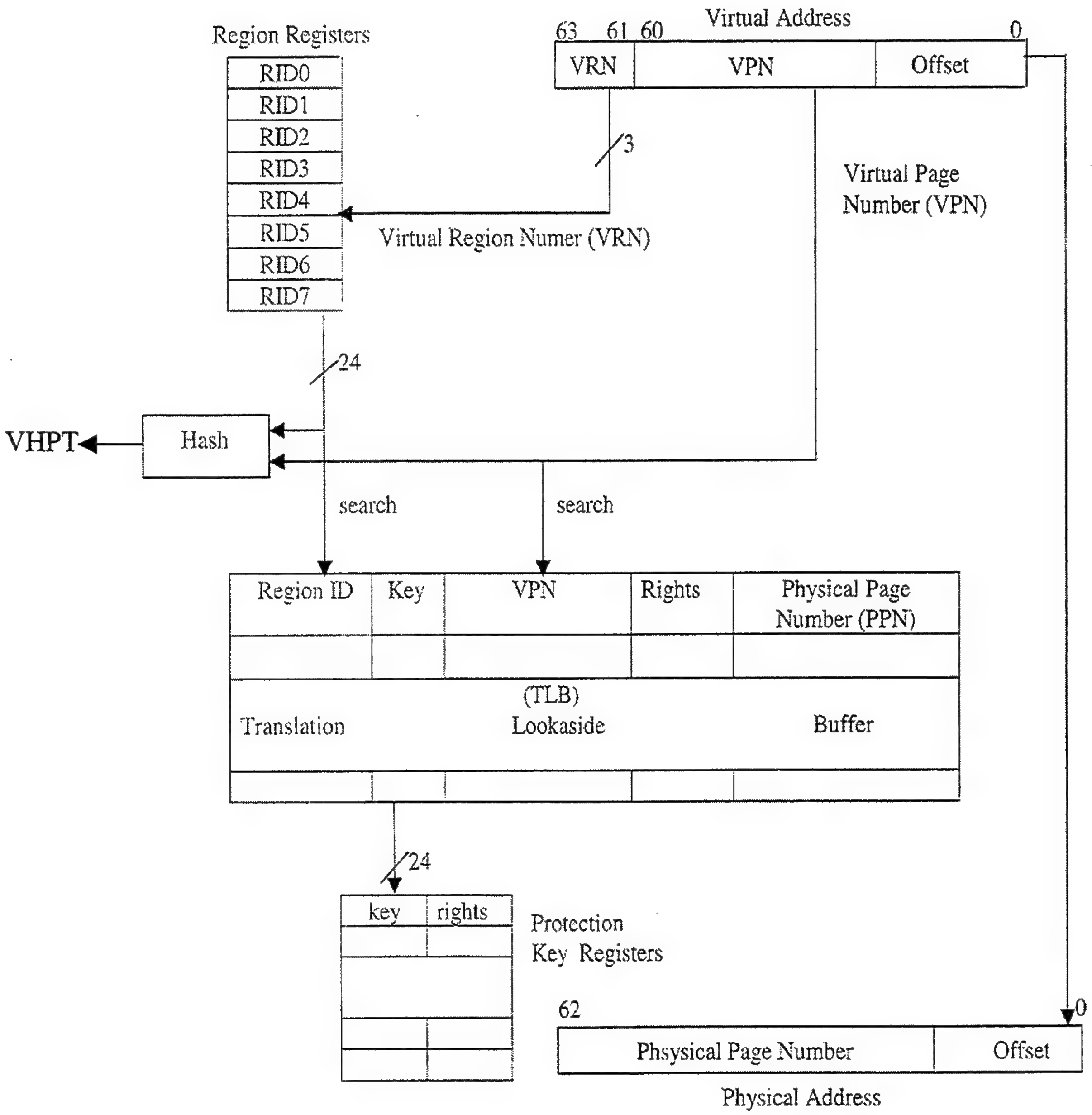

Figure 10. Virtual Address Translation From Ref. [2].

On each memory reference, the current privilege level field in the processor status register (PSR.cpl) is compared with the privilege level field of the page's TLB entry (TLB.pl). If the PSR.cpl privilege level is less than or equal to that of the TLB.pl, access is granted.

The third protection mechanism, access rights, determines the read, write and execute permissions for the page. If access is granted to the page (i.e., PSR.cpl <= TLB.pl), the access rights field of the TLB entry (TLB.ar) define the read, write and execute permissions to the page.

These three protection mechanisms, discussed so far, are applied to MAS systems. In addition to addressability, privilege level and access rights protection mechanisms, the IA-64 also provides protection domains. The protection domain mechanism is a critical feature of SAS systems. In a single global address space, different protection domains can be created by enabling protection key registers (PSR.pk=1). On a memory reference, a key in the TLB entry associated with the referenced page is compared with the keys in the 16 protection key registers associated with the process. If a match is found, access is granted to the page as long as it is of a type (read, write and execute) which is allowed by the wd, rd, xd fields of protection key registers. Since the operating system controls the contents of the protection key registers, it can change the access privileges of all the pages in a domain by changing the contents of any protection key register.

3. Memory Attributes

When virtual addressing is enabled, the memory attributes field in the TLB (TLB.ma) defines the speculative, cacheability, coalescing, coherency and cache write

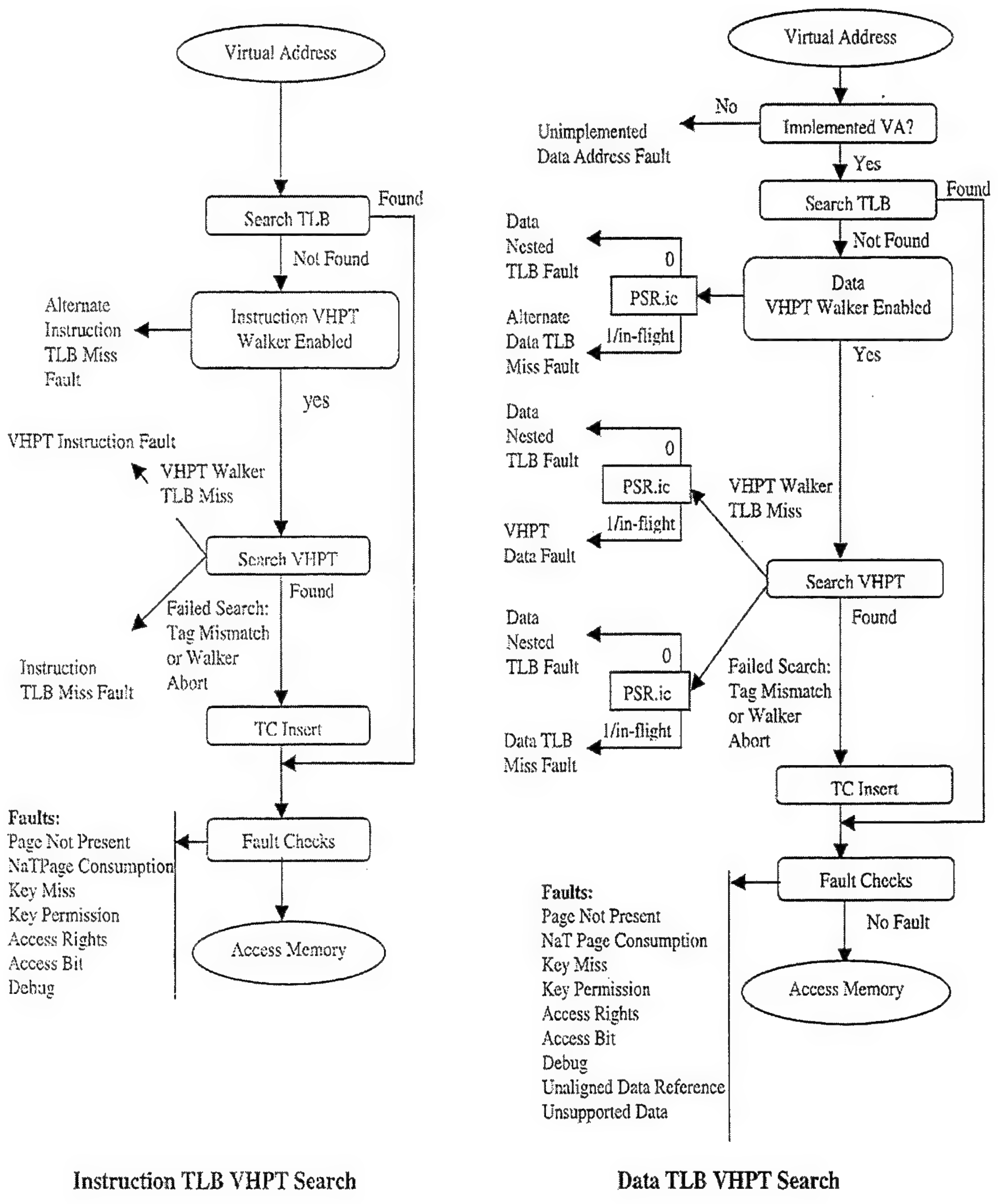

Figure 11. TLB/VHPT Search From Ref. [2].

policy attributes of the page. If a page is speculative, the processor performs load accesses without taking into account prior control dependencies. A page can be cacheable or uncacheable. If it is cacheable, the processor can copy the corresponding physical memory in all levels of the processor memory or cache hierarchy.

Write back cacheable pages must only modify the processor's copy of the physical memory, written data must be passed to the memory when the processor's copy is displaced. The coalescing attribute for uncacheable pages states that, multiple stores to this page may be gathered in a coalescing buffer and issued later as a single store operation.

When physical addressing is enabled, the 63rd bit of the address defines the memory attributes of the page. If the 63rd bit is zero, memory attributes of the page are cacheable, write-back and limited speculation. If it is one, memory attributes of the page are uncached, non-coalescing and non-speculative.

E. CONTEXT MANAGEMENT

Intel's IA-64 manual [Ref. 2] states context management and state preservation rules as described below.

1. Procedure Calls

Register preservation between procedures occurs in a way that,

- The caller saves GRs 2-3, GRs 8-11, GRs 14-15, Grs 16-31, FRs 6-15 and FRs 32-127.
- The callee saves GRs 4-7, FRs 2-5 and FRs 16-31
- GRs 32-127 are preserved by the Register Stack Engine (RSE)

- GR 0 is always zero. FR 0 is always +0.0 and FR1 is always +1.0.
- GR 1, GR 12 and GR13 have special uses.

2. Thread Switches Within The Same Address Space

When the operating system switches threads within the same address space, the following steps are taken.

- GRs, FRs, PRs, BRs and ARs are saved and restored.
- Memory ordering operations are performed. Memory ordering must satisfy read-after-write (RAW), write-after-read (WAR), and write-after-write (WAW) data dependencies to the same memory location.

3. Address Space Switching

When the operating system switches the address space, the following steps are taken.

- GRs, FRs, PRs, BRs and ARs are saved and restored.
- Memory ordering operations are performed. Memory ordering must satisfy read-after-write (RAW), write-after-read (WAR), and write-after-write (WAW) data dependencies to the same memory location.
- PKRs of the outbound address space are saved and then invalidated.
- DCRs of the outbound address space are saved.
- RRs of the outbound address space are saved.
- RRs of the inbound address space are restored.
- DCRs of the inbound address space are restored.

- PKRs of the inbound address space are restored.

F. IA-32 APPLICATIONS IN IA-64 SYSTEM ENVIRONMENT

The IA-64 architecture supports an IA-32 system environment. At any time, the processor can execute either IA-64 or IA-32 instructions. When the instruction set field of a processor status register (PSR.is) is zero, the IA-64 instruction set is executing. When this field is set to one, the IA-32 instruction set is executing.

Virtual addresses within both IA-64 and IA-32 instruction set can point the same physical memory location. While IA-64 instructions directly generate 64-bit virtual addresses, IA-32 instructions generate 16 or 32-bit effective addresses. These effective addresses are converted into 32-bit virtual addresses by IA-32 segmentation. Then, 32-bit virtual addresses are converted into 64-bit virtual addresses by zero extention. Finally, IA-64 memory management mechanisms translate virtual addresses generated by either instruction set into physical addresses.

The IA-64 architecture defines three special instructions and interruptions which give a transition between IA-64 and IA-32 instruction sets.

- A *jmpe* instruction is an IA-32 instruction that changes the instruction set to IA-64.
- A *br.ia* instruction is an IA-64 instruction that changes the instruction set to IA-32.
- *rfi* (return from interruption) instruction is used to return to IA-64 or IA-32 instruction set after handling an interrupt.

- All interrupts change the instruction set to the IA-64 instruction set for handling of interrupts.

IA-64 Reg	IA-32 Reg	Convention	Size	Description
General Purpose Integer Registers				
GR0				constant 0
GR1-3		undefined		scratch for IA-32 execution
GR4-7		unmodified		IA-64 preserved registers
GR8	EAX	IA-32 state	32	IA-32 general purpose registers
GR9	ECX			
GR10	EDX			
GR11	EBX			
GR12	ESP			
GR13	EBP			
GR14	ESI			
GR15	EDI			
GR16{15:0}	DS		64	IA-32 selectors
GR16{31:16}	ES			
GR16{47:32}	FS			
GR16{63:48}	GS			
GR17{15:0}	CS			
GR17{31:16}	SS			
GR17{47:32}	LDT			
GR17{63:48}	TSS			
GR18-23		undefined		scratch for IA-32 execution
GR24	ESD	IA-32 state	64	IA-32 segment descriptors (register format)
GR25-26		undefined		scratch for IA-32 execution
GR27	DSD	IA-32 state	64	IA-32 segment descriptors (register format)
GR28	FSD			
GR29	GSD			
GR30	LDTD			
GR31	GDTD			

GR32-127		undefined		IA-32 code execution space
Process Environment				
IP	IP	shared	64	Shared IA-32 and IA-64 virtual instruction pointer
Floating-point Registers				
FR0				constant +0.0
FR1				constant +1.0
FR2-5		unmodified		IA-64 preserved registers
FR6-7		undefined		IA-32 code execution space
FR8	MM0/FP0	IA-32 state	64/80	IA-32 MMX technology registers (aliased on 64-bit FP mantissa) IA-32 FP registers (physical registers vmapping)
FR9	MM1/FP1			
FR10	MM2/FP2			
FR11	MM3/FP3			
FR12	MM4/FP4			
FR13	MM5/FP5			
FR14	MM6/FP6			
FR15	MM7/FP7			
FR16-17	XMM0	IA-32 state	64	IA-32 streaming SIMD Extension registers low order 64-bits of XMM0 are mapped to FR16{63:0} High order 64-bits of XMM0 are mapped to FR17{63:0}
FR18-19	XMM1			
FR20-21	XMM2			
FR22-23	XMM3			
FR24-25	XMM4			
FR26-27	XMM5			
FR28-29	XMM6			
FR30-31	XMM7			
FR32-127		undefined		IA-32 code execution space
Predicate Registers				
PR0				constant 1
PR1-63		undefined		IA-32 code execution space
Branch Registers				
BR0-5		unmodified		IA-64 preserved registers
BR6-7		undefined		IA-32 code execution space

Application Registers				
RSC		unmodified		not used for IA-32 execution, IA-64 preserved registers
BSP				
BSPSTORE				
RNAT				
CCV		undefined	64	IA-32 code execution space
UNAT		unmodified		not used for IA execution, IA-64 preserved register
FPSR.sf0		unmodified		IA-64 numeric status and controls
FPSR.sf1,2,3		undefined		IA-32 code execution space
FSR	FSW,FTW, MXCSR	IA-32 state	64	IA-32 numeric status and tag word and Streaming SIMD Extension status
FCR	FCW,MXCSR		64	IA-32 numeric and Streaming SIMD Extension control
FIR	FOP,FIP,FCS		64	IA-32 x87 numeric environment opcode, code selector and IP
FDR	FEA,FDS		64	IA-32 x87 numeric environment data selector and offset
ITC	TSC	shared	64	Shared IA-32 time stamp counter (TSC) and IA-64 Internal Timer
PFS		unmodified		Not used for IA-32 code execution, prior EC is preserved in PFM IA-64 preserved registers
LC				
EC				
EFLAG	EFLAG	IA-32 state	32	IA-32 System/Arithmetic flags, writes of some bits condition by CPL and EFLAG.iopl.
CSD	CSD		64	IA-32 code segment (register format)
SSD	SSD			IA-32 code segment (register format)
CFLG	CR0/CR4		64	IA-32 control flags CR0=CFLG{31:0},CR4=CFLG{63:32}, Writable at CPL=0 only.

Table 2. IA-32 Application Register Mapping From Ref. [1]

IA-64 instructions can use all IA-64 and IA-32 application registers. (The mapping of an IA-32 register set to an IA-64 register set is shown in Table 2).

When executing IA-64 instructions, segmentation is disabled, and only virtual addresses are used. When executing IA-32 instructions within the IA-64 system environment, only IA-32 application registers are visible; IA-64 application and control registers cannot be accessed. Mapping of virtual addresses to physical addresses is accomplished through IA-64 memory management and I/O port references.

After analyzing the IA-64 architecture, now it will be convenient to introduce the virtual machine and virtual machine monitor concepts.

III. VIRTUAL MACHINE MONITORS

Rapid technological growth have transformed computer systems from single purpose systems into today's multi-access, multi-programming, multi-processing systems. Hence, this is a need for a single powerful computer system that can simulate multiple computers with different operating systems. In today's computer world, virtual machines have been used to address this need by providing multi-environment systems.

This chapter will discuss virtual machines (VMs) and virtual machine monitors (VMMs), the characteristics and logical modules of a VMM, and will give information about each type of VMM. Also, hardware and software requirements for each type of VMM will be given.

A. WHAT ARE VIRTUAL MACHINES?

Simply defined, a virtual machine is an identical and efficient copy of a real machine. Each VM can be considered a separate operating system platform.

There are many benefits to using a VM while making a program think that it is running on a "real machine." First, it is possible to execute privileged programs step by step on a virtual machine. Thus, virtual machines provide an environment for interactive debugging of an operating system. Current real machines seldom have an operator console thru which to examine the system state but this facility can be constructed relatively easily using a virtual machine monitor.

Second, virtual machines allow users to test new versions of an operating system. New versions of an operating system running on a virtual machine can be tested separately. Therefore, a user can test the new versions of an operating system but not have to completely reboot entire system from scratch.

Third, virtual machines can be used to upgrade an operating system and its applications. While running the original operating system in a virtual machine exactly as it did on the real machine, the upgraded operating system runs in a separate virtual machine. This prevents the corruption of the original operating system.

Fourth, virtual machines allow users to run programs in a virtual environment that may have different environment configurations from those of the real machine (i.e., the amount of memory or the input-output devices attached). This allows a user to run applications on different environments without having to use another real machine.

Finally, virtual machines allow multiple operating systems to be run on the same computer concurrently. For example, a user can run Windows 98, Windows NT, Linux etc. concurrently on the same computer. This allows users to develop applications for different operating systems much more easily. Since several operating systems may run on the same computer concurrently, a user can test his applications on many operating systems, using just one computer.

B. PURPOSE OF A VMM

A VMM is a software program that mediates between the virtual machine and the real machine. The VMM manages the real resources and allocates these resources to the VMs as needed. Goldberg defines a VMM as, "a hardware-software duplicate of a real existing computer system in which a statistically dominant subset of the virtual processor's instructions execute on the host processor." [Ref. 7].

1. Characteristics of VMM

A VMM has three essential characteristics [Ref. 8]. First, the VMM provides an environment for the VMs which is identical to the real environment. Here, the real environment determines a complete computer system. There may be some differences between the VM's environment and real environment caused by the timing dependencies and the availability of system resources.

Second, a VMM is efficient in that the real processor executes a statistically dominant subset of the virtual processor's instructions without VMM interpretation. A VMM simulates the effect of only those instructions which trap.

A third characteristic is that a VMM has complete control of the real system resources. For a virtual processor, it is only possible to access real system resources if the real system resources are explicitly allocated to the virtual processor by the VMM. In order to prevent a virtual machine from taking control of the real machine, the virtual machine must be run in user mode, and must not be permitted to put the real machine into a mode which allows the virtual machine to execute privileged instructions. Once the VMM allocates the resources to a virtual processor, it is then possible for the VMM to regain control of these allocated resources. So, the VMM will be able to allocate real system resources to other VMs as well, when needed.

2. Modules of a VMM

A VMM consists of three sets of modules, a dispatcher, an allocator, and an interpreter. The dispatcher is the top-level control module that decides which module to call when a trap occurs. When a VM attempts to execute a privileged instruction, the VM traps to the dispatcher, and the dispatcher invokes the allocator. An allocator is the second module that controls the system resources. It is the allocator's responsibility to avoid giving the same resources to more than one virtual processor simultaneously. The third in the set of modules is the interpreter. Each privileged instruction has an interpreter that is used to simulate the effect of the instruction that caused a trap.

3. Types of VMMs

There are three types of VMMs: Type I VMM, Type II VMM, and Hybrid VMM (HVM).

Every process contains instructions that will be executed on hardware. The ratio of the number of instructions that execute directly on hardware to those that must be emulated in software determines if a machine is a Complete Software Interpreter Machine (CSIM), a real machine, a VMM or an HVM. All of the CSIM instructions execute by software interpretation, no CSIM instruction executes directly on a real processor. Therefore, a CSIM is not a VMM. Examples of CSIM include the Java byte code interpreter [Ref. 9] and the Transmeta Crusoe chip which provides binary code translation from one instruction set to another [Ref. 10]. Nor is a real machine a VMM since all the instructions execute directly on the real hardware, and no software interpretation is required. Although, the execution of a statistically dominant subset of the virtual processor's instructions on a real machine is an implementation requirement for VMMs, exactly how the VMM takes control of the instructions to be interpreted is not specified and can vary from one type of VMM to another. For example, a Type I VMM and a Type II VMM are distinguished from each other according to how the VMM gains control of the instructions to be interpreted.

A Type I VMM runs directly on a real machine. Figure 12 (below) shows the structure of a Type I VMM. A Type I VMM can be treated as an operating system that is responsible for providing an execution environment and resource allocation for VMs. Since a Type I VMM is responsible for resource allocation and provides an execution environment, this type of VMM may be more complex, and may include more code than a Type II VMM. For a Type I VMM, when sensitive, privileged instructions are executed in a VM, they cause a trap to the VMM. Then, the VMM will execute a code that emulates the behavior of the privileged instruction for the VM.

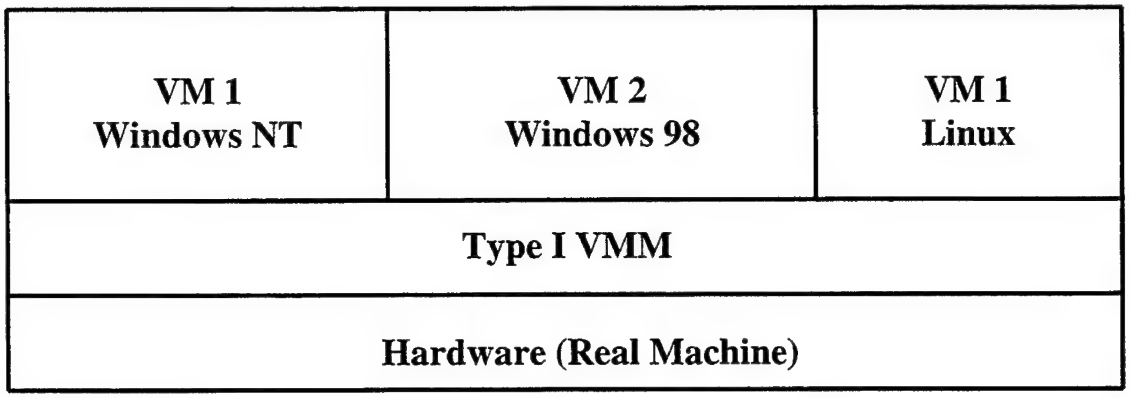

Figure 12. Illustration of a Type I VMM

In contrast, a Type II VMM runs as an application under a host operating system. In other words, a Type II VMM runs as an extended machine and only provides virtualization services. A Type II VMM is illustrated in Figure 13. The host operating system handles the resource allocation and provides the extended environment for the VMM. When sensitive, privileged instructions are executed in a VM, they cause a trap to the host operating system. Then the host operating system gives control to the VMM services, that will execute code to emulate the behavior of the privileged instruction for the VM.

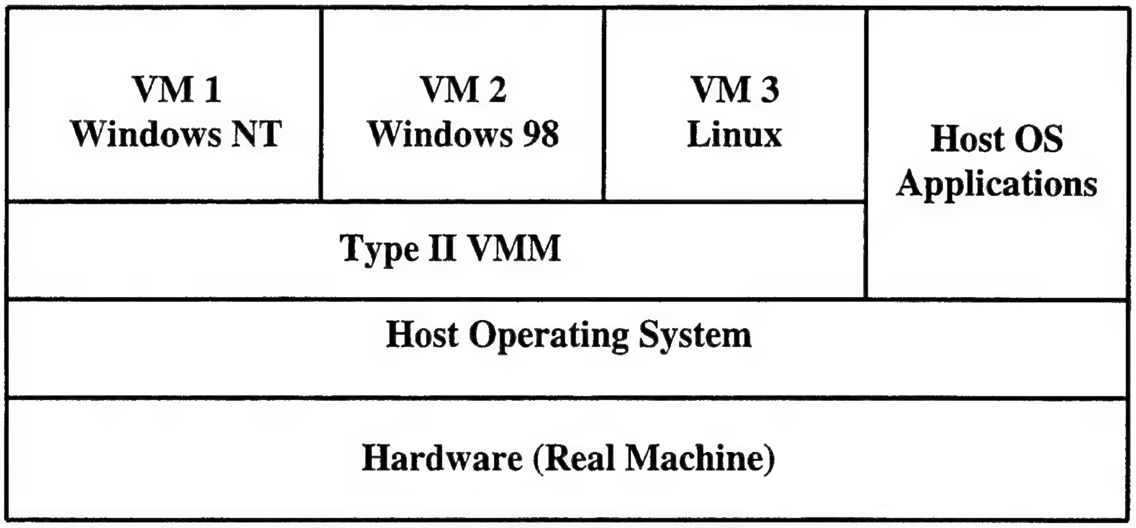

Figure 13. Illustration of a Type II VMM

An Hybrid Virtual Machine (HVM) is identical to a VMM. The major differences between an HVM and a VMM are that in a HVM, all privileged instructions are interpreted by software, while in a VMM, privileged instructions can execute directly on the real processor.

C. HARDWARE & SOFTWARE REQUIREMENTS OF VMMS FOR VIRTUALIZATION

In the previous sections, three types of VMM are defined. In the following sections, hardware and software requirements for each type of VMM will be discussed.

A processor's job is to execute instructions. Since the real processor and the virtual processor are identical, some of the virtual machine's instructions can be executed directly on the real processor without VMM interpretation. However, some virtual machine instructions cannot be executed directly on the real processor. These instructions are called *sensitive instructions* [Ref. 7]. When sensitive instructions are executed, the state of the real processor is changed, sensitive registers as well as sensitive memory locations are essentially altered. Therefore, sensitive instructions can cause interpretation problems and interfere with the underlying VMM if they are executed directly by a VM. Clearly, the key to implementing a VMM is to prevent the execution of sensitive instructions directly on the real processor.

Some Intel IA-32 and IA-64 instructions are privileged, meaning that, if they are not executed in privilege level zero, they cause a general protection exception. In general, a VMM is executed in supervisor mode, and a VM is executed in user mode. When a VM attempts to execute a privileged instruction, a trap to the VMM occurs. In this manner, the VMM can simulate the effect of the instruction that caused the trap and then return control to the VM.

In order to support a VMM, a processor must be able to be virtualized. If all the sensitive instructions of a processor are also privileged, then the processor is said to be "virtualizable". This means all sensitive, privileged instructions will trap to the VMM. After trapping, the VMM will simulate the effect of the sensitive, privileged instruction for the VM. On the other hand, sensitive, non-privileged instructions may cause interference with the proper operation of multiple VMs. So, the VMM must examine each instruction before execution to ensure that it is not a sensitive, non-privileged instruction. When the VMM encounters a sensitive instruction, the VM must be forced to trap to the VMM.

In the following sections, the hardware and software requirements for virtualization will be analyzed for each type of VMM.

1. Hardware Requirements for a Type I VMM

As discussed earlier, a Type I VMM runs on a real machine and performs resource allocation and systems scheduling. Goldberg developed three hardware requirements for virtualization in order to implement a Type I VMM on a processor's hardware [Ref. 7]. These three requirements are:

(1) For a large subset of instructions, the method of executing non-privileged instructions in both supervisor and user modes must be roughly equivalent.

(2) There must be a mechanism, such as a protection system or address translation system that protects the real machine and virtual machines from the active VM.

(3) There must be a method of automatically signaling the VMM when the virtual machine attempts to execute a sensitive instruction. Then, it must be possible for the VMM to simulate the effect of this instruction. Sensitive instructions include:

(a) Those instructions that change or reference the mode of the virtual machine or the state of the machine.

(b) Those instructions that change or read the sensitive registers or memory locations.

(c) Those instructions that reference the storage protection system or address relocation system.

(d) All input-output instructions.

2. Software Requirements for a Type II VMM

Recall that a Type II VMM runs under a host operating system rather than on a real machine. Since the host operating system must provide primitives that make it possible to implement a Type II VMM on a processor's hardware, Goldberg introduced three software requirements for a host operating system that will complement the hardware requirements and must be met in order to build a Type II VMM on a processor [Ref. 7]. These software requirements are:

(1) The host operating system cannot invalidate the requirement that specifies that for a large subset of instructions, the method of executing non-privileged instructions in both supervisor mode and user mode must be roughly equivalent.

The hardware requirement 1 is still applicable for the host operating system and is not affected by the insertion of an operating system between a VMM and hardware, and is therefore still needed.

(2) There must be a primitive available in the host operating system, such as an address translation primitive, a protection primitive or a sub-process primitive to be able to protect the VMM and other virtual machines from the active VM.

The hardware requirement 2 for Type I VMM is modified since the VMM will be running on an extended and insulated machine controlled by the host operating system that, likely, will prevent the VMM from utilizing a protection and address translation system. Therefore, there must be a system primitive that can be invoked by the VMM to simulate the effects of sensitive, privileged instructions.

(3) There must be a primitive available in the host operating system to direct the signal from the host operating system to the VMM when a VM attempts to execute a sensitive instruction. Then, it must be possible for the VMM to simulate the effect of the sensitive instruction.

Since the VMM and the host operating system are two separate system components, hardware requirement 3 for Type I VMM is modified so that all of the VM traps will be directed from the host operating system to the VMM.

3. Hybrid VMM

The Hybrid VMM (HVM) executes all non-privileged instructions directly on the processor while interpreting all privileged instructions in software.

The hardware requirements for a Type I VMM are changed to support a virtualizable HVM such that,

(1) Requirement 1, (i.e, for a large subset of instructions, the method of executing non-privileged instructions in both supervisor mode and user mode must be roughly equivalent), is eliminated.

(2) Requirement 3a, (i.e., there must be a method of automatically signaling the VMM when the virtual machine attempts to execute a instruction which changes or references the mode of the virtual machine or the state of the machine) is weakened. Only unprivileged instructions which change or reference the mode of the virtual machine or the state of the machine must be identified since all privileged instructions are interpreted by software.

The other hardware requirements for a Type I VMM are still valid for an HVM.

This thesis will focus on the analysis of the Intel IA-64 processor support for a secure VMM or HVM. Previous analyses have indicated that, a secure implementation of a VMM will likely be a Type I VMM or HVM [Ref. 7]. This thesis will not analyze the software requirements for the host operating system for implementing a VMM.

This chapter defined each type of VMM and their hardware/software requirements for virtualization. Now, we will move on to analyze the Intel IA-64 architecture and instruction set to see if it is able to support any type of VMM.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. THE INTEL IA-64 ARCHITECTURE'S CAPABILITY TO SUPPORT A VIRTUAL MACHINE MONITOR

This chapter will analyze the capacity of Intel IA-64 architecture to support a virtual machine monitor by using the hardware requirements described in Chapter III. Each member of the Intel IA-64 instruction set will be examined to find whether or not it supports virtualization.

Like most of the other processors, the IA-64 architecture was not designed to support virtual machines. Therefore, whether or not the IA-64 architecture supports a virtual machine monitor is a "hit-or-miss proposition".

The hardware requirements for virtualization are given in Chapter III. Recall that any instruction that violates the hardware requirements 1, 2, 3a, 3b, 3c or 3d prevents the processor from running a Type I VMM or a Type II VMM. Also, any instruction that violates the hardware requirements 2, 3a (weaker form), 3b, 3c or 3d prevents the processor from running a HVM. As such, any instruction that violates the hardware requirements 2, 3a (weaker form), 3b, 3c or 3d makes a processor non-virtualizable.

Before examining each IA-64 instruction, the IA-64 architecture will be analyzed to see if it meets three main requirements. The first requirement states that for a large subset of the instructions, the method of executing non-privileged instructions in both supervisor and user modes must be roughly equivalent. The IA-64 architecture meets this requirement because there is no difference between the methods of executing instructions in supervisor and user modes.

The second requirement states that there must be a mechanism, such as a protection system or address translation system that protects the real and virtual machines from the active VM. IA-64 architecture meets this requirement. As described in Chapter II, the IA-64 architecture defines virtual regions to support both MAS and SAS models. Each processor implements 2^{24} virtual regions and each process can have up to eight regions. The operating system maps these regions through region identifiers, loaded into region registers before the process executes.

A process cannot reference the regions of other processes unless the operating system allows shared access to that region. This mechanism is the addressability protection mechanism. Privilege level and access rights are the other two protection mechanisms. On each memory access, PSR.cpl is compared with the TLB.pl. If PSR.cpl <= TLB.pl, access is allowed to the page. When access is granted to the page, the access rights field of the TLB entry (TLB.ar) defines the read, write and execute permissions to the page.

In addition to addressability, privilege level and access rights protection mechanisms, the IA-64 architecture also provides protection domains. In a single global address space, different protection domains can be created by enabling protection key registers (PSR.pk=1). On a memory reference, a key in the TLB entry associated with the referenced page is compared with the keys in the 16 protection key registers. If a match is found, access is granted to the page as long as it is of a type (read, write and execute) which is allowed by the wd, rd, xd fields of protection key registers. While the access rights bits associated with each translation provide privilege level-granular access to a page, protection keys permit domain-granular access to a page.

The third requirement states that there must be a method of automatically signaling the VMM when the virtual machine attempts to execute a sensitive instruction.

Then, it must be possible for the VMM to simulate the effect of this instruction. The IA-64 architecture uses interrupts, faults and traps that cause the flow of control to be passed to an interrupt handling routine when an unprivileged task executes a privileged instruction. However, the IA-64 instruction set contains sensitive, unprivileged instructions. Since the unprivileged instructions do not generate an interrupt, fault or trap, the VMM will not simulate the effect of the instruction. These instructions prevent the IA-64 architecture from being virtualizable.

In the following section, each of the IA-64 architecture's instructions (documented as of July 2000 [Ref. 3]) will be examined, and those that violate hardware virtualization requirements will be discussed in detail.

A. EXAMINATION OF THE IA-64 INSTRUCTION SET

Table 3 (below) contains all of the IA-64 instructions found in the Intel IA-64 instruction set reference [Ref. 3]. The CODE column shows the name of the instruction. The DESCRIPTION column gives the short description of the instruction, and the SENS column indicates whether the instruction is sensitive. The PRIV column indicates whether the instruction is privileged. The PROB column shows whether the instruction prevents virtualization, and the REAS column indicates why the instruction causes a problem for virtualization. The meaning of the abbreviations used in REAS column is provided below.

3a1	: References or changes the mode of the processor.
3a2	: References or changes the state of the processor.
3b1	: References or changes sensitive registers.

3b2 : References or changes sensitive memory locations.

3c1 : References or changes privilege level.

3c2 : References the memory system.

3c3 : Interferes with address translations.

3d : I/O instructions.

CODE	DESCRIPTION	SENS	PRIV	PROB	REAS
add	Add	N	N	N	-
addp4	Add Pointer	N	N	N	-
alloc	Allocate Stack Frame	N	N	N	-
and	Logical And	N	N	N	-
andcm	And Complement	N	N	N	-
br.cloop br.ctop br.cexit br.wtop br.wexit	Loop Branch	N	N	N	-
br.call	Procedure Call	Y	N	Y	3c1
br.ret	Procedure Return	Y	N	Y	3c1
br.ia	Invoke IA-32 Instruction Set	Y	N	Y	3a1
break	Break	N	N	N	-
brp	Branch Predict	N	N	N	-
bsw	Bank Switch	Y	Y	N	-
chk	Speculation Check	N	N	N	-
clrrrb	Clear RRB	N	N	N	-
cmp	Compare	N	N	N	-
cmp4	Compare Word	N	N	N	-
cmpxchg	Compare and Exchange	Y	N	Y	3c1
cover	Cover Stack Frame	Y	N	Y	3b1
czx	Compute Zero Index	N	N	N	-
dep	Deposit	N	N	N	-
epc	Enter Privileged Code	N	N	N	-
extr	Extract	N	N	N	-
fabs	Floating-point Absolute Value	N	N	N	-
fadd	Floating-point Add	N	N	N	-
famax	Floating-point Absolute Maximum	N	N	N	-
famin	Floating-point Absolute Minimum	N	N	N	-
fand	Floating-point Logical And	N	N	N	-

fandcm	Floating-point And Complement	N	N	N	-
fc	Flush Cache	Y	N	Y	3c1 3c2
fchkf	Floating-point Check Flags	N	N	N	-
fclass	Floating-point Class	N	N	N	-
fclrf	Floating-point Clear Flags	N	N	N	-
fcmp	Floating-point Compare	N	N	N	-
fcvt.fx	Convert Floating-point to Integer	N	N	N	-
fcvt.xf	Convert Signed Integer to Floating-point	N	N	N	-
fcvt.xuf	Convert Unsigned Integer to Floating-point	N	N	N	-
fetchadd	Fetch and Add Immediate	Y	N	Y	3c1
flushrs	Flush Register Stack	N	N	N	-
fma	Floating-point Multiply Add	N	N	N	-
fmax	Floating-point Maximum	N	N	N	-
fmerge	Floating-point Merge	N	N	N	-
fmin	Floating-point Minimum	N	N	N	-
fmix	Floating-point Mix	N	N	N	-
fmpy	Floating-point Multiply	N	N	N	-
fms	Floating-point Multiply Subtract	N	N	N	-
fneg	Floating-point Negate	N	N	N	-
fnegabs	Floating-point Negate Absolute Value	N	N	N	-
fnma	Floating-point Negative Multiply Add	N	N	N	-
fnmpy	Floating-point Negative Multiply	N	N	N	-
fnorm	Floating-point Normalize	N	N	N	-
for	Floating-point Logical Or	N	N	N	-
fpabs	Floating-point Parallel Absolute Value	N	N	N	-
fpack	Floating-point Pack	N	N	N	-
fpamax	Floating-point Parallel Absolute Maximum	N	N	N	-
fpamin	Floating-point Parallel Absolute Minimum	N	N	N	-
fpcmp	Floating-point Parallel Compare	N	N	N	-
fpcvt.fx	Convert Parallel Floating-point to Integer	N	N	N	-

fpma	Floating-point Parallel Multiply Add	N	N	N	-
fpmax	Floating-point Parallel Maximum	N	N	N	-
fpmerge	Floating-point Parallel Merge	N	N	N	-
fpmin	Floating-point Parallel Minimum	N	N	N	-
fpmpy	Floating-point Parallel Multiply	N	N	N	-
fpms	Floating-point Parallel Multiply Subtract	N	N	N	-
fpneg	Floating-point Parallel Negate	N	N	N	-
fpnegabs	Floating-point Parallel Negate Absolute Value	N	N	N	-
fpnma	Floating-point Parallel Negative Multiply Add	N	N	N	-
fpnmpy	Floating-point Parallel Negative Multiply	N	N	N	-
fprcpa	Floating-point Parallel Reciprocal Approximation	N	N	N	-
fprsqrta	Floating-point Parallel Reciprocal Square Root Approximation	N	N	N	-
frcpa	Floating-point Reciprocal Approximation	N	N	N	-
frsqrta	Floating-point Reciprocal Square Root Approximation	N	N	N	-
fselect	Floating-point Select	N	N	N	-
fsetc	Floating-point Set Controls	N	N	N	-
fsub	Floating-point Subtract	N	N	N	-
fswap	Floating-point Swap	N	N	N	-
fsxt	Floating-point Sign Extend	N	N	N	-
fwb	Flush Write Buffers	N	N	N	-
fxor	Floating-point Exclusive Or	N	N	N	-
getf	Get Floating-point Value or Exponent or Significand	N	N	N	-
invala	Invalidate ALAT	N	N	N	-
itc	Insert Translation Cache	Y	Y	N	-
itr	Insert Translation Register	Y	Y	N	-
ld	Load	Y	N	Y	3c1
ldf	Floating-point Load	Y	N	Y	3c1
ldfp	Floating-point Load Pair	Y	N	Y	3c1
lfetch	Line Prefetch	Y	N	Y	3c1 3c2
loadrs	Load Register Stack	N	N	N	-

mf	Memory Fence	N	N	N	-
mix	Mix	N	N	N	-
mov ar	Move Application Register	Y	Y	N	-
mov br	Move Branch Register	N	N	N	-
mov cr	Move Control Register	Y	Y	N	-
mov fr	Move Floating-point Register	N	N	N	-
mov gr	Move General Register	N	N	N	-
mov imm	Move Immediate	N	N	N	-
mov indirect	Move Indirect Register	Y	Y	N	-
mov ip	Move Instruction Pointer	N	N	N	-
mov pr	Move Predicates	N	N	N	-
mov psr	Move Processor Status Register	Y	Y	N	-
mov um	Move User Mask	Y	N	Y	3b1
movl	Move Long Immediate	N	N	N	-
mux	Mux	N	N	N	-
nop	No Operation	N	N	N	-
or	Logical Or	N	N	N	-
pack	Pack	N	N	N	-
padd	Parallel Add	N	N	N	-
pavg	Parallel Average	N	N	N	-
pavgsub	Parallel Average Subtract	N	N	N	-
pcmp	Parallel Compare	N	N	N	-
pmax	Parallel Maximum	N	N	N	-
pmin	Parallel Minimum	N	N	N	-
pmpy	Parallel Multiply	N	N	N	-
pmpyshr	Parallel Multiply and Shift Right	N	N	N	-
popcnt	Population Count	N	N	N	-
probe	Probe Access	Y	N	Y	3c1
psad	Parallel Sum of Absolute Difference	N	N	N	-
pshl	Parallel Shift Left	N	N	N	-
pshladd	Parallel Shift Left and Add	N	N	N	-
pshr	Parallel Shift Right	N	N	N	-
pshradd	Parallel Shift Right and Add	N	N	N	-
psub	Parallel Subtract	N	N	N	-
ptc.e	Purge Translation Cache Entry	Y	Y	N	-
ptc.g, ptc.g	Purge Global Translation Cache	Y	Y	N	-
ptc.l	Purge Local Translation Cache	Y	Y	N	-
ptr	Purge Translation Register	Y	Y	N	-
rfi	Return from Interrupt	Y	Y	N	-
rsm	Reset System Mask	Y	Y	N	-

rum	Reset User Mask	Y	N	Y	3b1
setf	Set Floating-point Value, Exponent or Significand	N	N	N	-
shl	Shift Left	N	N	N	-
shladd	Shift left and Add	N	N	N	-
sladdp4	Shift Left and Add Pointer	N	N	N	-
shr	Shift Right	N	N	N	-
shrp	Shift Right Pair	N	N	N	-
srlz	Serialize	N	N	N	-
ssm	Set System Mask	Y	Y	N	-
st	Store	Y	N	Y	3c1
stf	Floating-point Store	Y	N	Y	3c1
sub	Subtract	N	N	N	-
sum	Set User Mask	Y	N	Y	3b1
sxt	Sign Extend	N	N	N	-
sync	Memory Synchronization	N	N	N	-
tak	Translation Access Key	Y	Y	N	-
tbit	Test Bit	N	N	N	-
thash	Translation Hashed Entry Address	N	N	N	-
tnat	Test NaT	N	N	N	-
tpa	Translate to Physical Address	Y	Y	N	-
ttag	Translation Hashed Entry Tag	N	N	N	-
unpack	Unpack	N	N	N	-
xchg	Exchange	Y	N	Y	3c1
xma	Fixed-point Multiply Add	N	N	N	-
xmpy	Fixed-point Multiply	N	N	N	-
xor	Exclusive Or	N	N	N	-
zxt	Zero Extend	N	N	N	-

Table 3. IA-64 Instruction Set Analysis

B. INSTRUCTIONS THAT VIOLATE THE VIRTUALIZATION REQUIREMENTS

1. Branch Instructions

In the IA-64 architecture, branches can be either IP-relative or indirect. For IP-relative branches, the target address is encoded in the branch instruction as a signed-immediate displacement. For indirect branches, the target address is obtained from the r2 field of branch register (BR-r2).

IA-64 architecture has several branch types. These branch types are: *br.call, br.ret, br.ia, br.cloop, br.ctop, br.cexit, br.wtop, br.wexit.* The branch types that violate the virtualization requirements will be explained below.

a. br.call Instruction

When the *br.call* instruction executes, if the predicate register value is set, the branch is taken, otherwise branch is not taken. If the branch is taken several actions occur; the current values of the CFM, the EC and the current privilege level (cpl) are saved in the previous function state application register.

The *br.call* instruction prevents virtualization because this instruction references the privilege level in the system. Normally, the operating system of the virtual machine (VMOS) does not execute at privilege level zero. Rather, the operating system executes at the application level (privilege level three). Thus, all privileged instructions will cause traps to the VMM. However, operating systems assume that they are running at privilege level zero. This assumption may cause a problem if the VMOS executes the *br.call* instruction and a branch is taken. When the VMOS examines the cpl field in the processor status register to save the current privilege level into the previous function state register, the VMOS will find that its privilege level is not zero. This is a problem because the process may halt.

b. br.ret Instruction

The reason that the *br.ret* instruction prevents virtualization is similar to that explained in the previous paragraph. When the *br.ret* instruction executes, the CFM, the EC and the current privilege level are restored from the previous function state register. The privilege level is restored if the return does not lower the privilege level.

Since the VMOS assumes that it is running at privilege level zero, after restoring cpl and examining the privilege level, the process may halt upon finding that the VMOS is not operating at privilege level zero.

c. *br.ia Instruction*

The *br.ia* instruction invokes the IA-32 instruction set by setting the PSR.is field to one, and begins execution of IA-32 instructions at the virtual linear target address obtained from BR-2.

This instruction prevents virtualization because it changes the mode of the processor. While the processor is executing the IA-32 instruction set, only IA-32 application register file is visible to the processes. Table 2 in Chapter II shows the IA-32 application register mapping. IA-32 real, VM86 and protected mode segmentation is in effect. Segment protection checks and virtual address generation are applied according to IA-32 segmentation rules.

However, in the IA-64 system environment, user-level instructions can modify all descriptors and selectors including GDT and LDT. Modifications of the LDT and GDT cause problems, because, within the IA-32 system environment, the GDT and LDT are privileged operating system segmentation tables and can be accessed only at privilege level zero [Ref. 2].

Also, previous analyses have indicated that the IA-32 instruction set has seventeen instructions that violate the virtualization requirements [Ref. 11]. Therefore, switching the instruction set from IA-64 to IA-32 will provide a system environment in which these seventeen instructions can execute.

2. cmpxchg, fetchadd, xchg Instructions

The *cmpxchg* instruction reads a value from memory starting at the address specified by the value in GR-r3 field. The value is zero extended and then compared with contents of the compare and exchange value register (CCV). If the two are equal, then the least significant bytes of the value in GR-r2 are written to memory starting at the address contained in GR-r3. The zero extended value is placed in GR-r1, the NaT bit corresponding to GR-r1 is cleared.

The *xchg* instruction reads a value from memory starting at the address specified by the value in GR-r3. The least significant bytes of the value in GR-r2 are written to memory starting at the address contained in GR-r3. The zero extended value is placed in GR-r1, the NaT bit corresponding to GR-r1 is cleared.

The *fetchadd* instruction reads four or eight bytes from memory starting at the address specified by the value in GR-r3. The value is zero extended and added to the sign extended immediate value specified by the operand. The least significant four or eight bytes of the sum are written to memory starting at the address specified by the value in GR-r3. The zero extended value is placed in GR-r1, the NaT bit corresponding to GR-r1 is cleared.

Using the protection mechanisms for the IA-64 architecture described in Chapter II, all memory references are checked before allowing access. The TLB.ar, TLB.pl fields and privilege levels of the access (PSR.cpl) determine the IA-64 access to a page.

These instructions violate the hardware virtualization requirement because they all reference the privilege level in the system. When these instructions execute, both read and write access privileges are required for the referenced page. Recall that TLB.ar, TLB.pl and PSR.cpl fields defines read, write and execute accesses to a page. Consider the following scenario. A VM, running at privilege level three, wants to reference a memory address. To get the access rights for the referenced page, the PSR.cpl field will be examined. Then, the VM will find that it is not running at privilege level zero, at which it assumes to run. This may halt the process.

3. cover Instruction

This instruction allocates a new stack frame of zero size. If interrupt collection is enabled (PSR.ic=0), then the old value of the CFM is copied to the interrupt function state register (IFS).

This instruction is a problem for virtualization because it changes the contents of IFS, which is a sensitive interrupt register. IFS can only be accessed at privilege level zero. When a VM running at privilege level three executes this instruction, since this instruction is not privileged, a trap to the VMM will not occur and the VMM will not be able to simulate the behavior of this instruction for the VM.

4. probe Instruction

The *probe* instruction is used to determine whether read or write access to a given virtual address is allowed with the current privilege level. This instruction can also be used to probe with lower privilege level.

The *probe* instruction references the system privilege level. When the probe instruction executes, the current privilege level is referenced in order to determine read or write access to a given virtual address. Since a VM normally operates at privilege level three, privilege checks once again become a problem when the VM is not running at the highest privilege level at which the VMOS assumes itself to be running.

5. fc, lfetch Instructions

The IA-64 data memory hierarchy is composed of zero or more levels of cache between the memory and the register files.

The *fc* instruction invalidates the cache line that contains at least 32-bytes of data, associated with the address obtained from the value of GR-r3, from all levels of the processor data memory hierarchy.

The *lfetch* instruction moves a line of data (32-bytes or more) containing the address specified by the value in GR-r3 to the highest level of the data memory hierarchy.

These instructions prevent virtualization because they reference the memory system by invalidating the cache lines and moving the lines in the memory hierarchy. Also, these instructions reference privilege level. When executed at privilege level zero, the *fc* does not perform any access rights or protection key checks, but does perform access rights checks at other privilege levels. When the *lfetch* executes, the memory read operation is described by the memory attribute of the accessed page. Again, the privilege level checks may cause problems for the VMs that assume themselves to be running at privilege level zero.

6. ld, ldf, ldfp, st, stf Instructions

The *ld* instruction reads a value from memory starting at the address specified by the value in GR-r3. Then, the value is zero extended and placed in GR-r1. The *ldf* instruction reads a value from memory starting at the address specified by the value in GR-r3. Then, the value is converted into the floating-point register format and placed in FR-r1. The *ldfp* instruction reads eight or sixteen bytes from memory starting at the address specified by the value in GR-r3. By encoding the instruction, the value read is treated as integer/parallel, floating-point data or a contiguous pair of floating-point numbers. Each number is converted into the floating-point register format. The value at the highest address is placed in FR-r2, the value at the lowest address is placed in FR-r1.

These instructions inhibit virtualization because they all reference the privilege levels in the system. When these instructions execute, a memory reference check is performed. In order to load a value from memory, a process should have enough privilege and access rights to reference that page. If a VM running at privilege level three wants to load a value from memory, the privilege level of the access will be examined. Upon finding that the privilege level is not zero (i.e., the level which VMOS expects), the process may halt.

The *st* instruction writes the value in GR-r2 to memory, starting at the address specified by the value in GR-r3. On the other hand, the *stf* instruction writes the value in FR-r2 to memory, starting at the address specified by the value in GR-r3.

The reason that the *st* and *stf* instruction prevent virtualization is similar to that for the load operation. Again, when these instructions execute, a memory reference check is performed, and VMOS will find that it is not operating at the expected privilege level.

7. mov um, sum, rum Instructions

The *mov um* instruction has two forms: (1) the move from user mask form copies the zero-extended contents of the user mask (PSR {5:0}) into GR-r1; and (2) the move to user mask from copies the 5-bits of GR-r2 {5:0} into the user mask.

The *rum* instruction takes the complement of the immediate operand and performs a logical AND operation with the user mask. The result is placed in the user mask.

Conversely, the *sum* instruction performs a logical OR operation between the immediate operand and the user mask, and places the result in the user mask.

These instructions prevent virtualization because all of them reference sensitive registers. The user mask is a subset of the processor status register (PSR {5:0}), and controls memory access alignment, byte-ordering and user-configured performance monitors.

The contents of the user mask are not saved and restored at context switching. When a process executes one of these instructions, since these are sensitive but not privileged instructions, the VMM cannot simulate the behavior of this instruction for the VM. This will cause unexpected results for the VM.

C. CONCLUSION

After examining each instruction of the Intel IA-64 instruction set, 18 instructions (detailed in Table 5) are found to make the IA-64 architecture non-virtualizable.

INSTRUCTIONS	REASON OF VIOLATION							
	3a1	3a2	3b1	3b2	3c1	3c2	3c3	3d
br.call, br.ret, cmpxchg, probe, fetchadd, xchg, ld, ldf, ldfp, st, stf					X			
fc, lfetch					X	X		
cover, mov um, rum, sum			X					
br.ia	X							

Table 4. Instructions That Violate Hardware Requirements

This analysis indicates that the Intel IA-64 architecture is not virtualizable according to the hardware requirements introduced by Goldberg.

THIS PAGE INTENTIONALLY LEFT BLANK

V. COMPARISON WITH OTHER PROCESSORS

Analysis of the IA-64 instruction set has shown that the IA-64 architecture is not virtualizable. However, some virtualization techniques can be used to virtualize the IA-64 architecture.

This chapter will discuss several architectures and virtualization techniques that might be applicable to virtualizing the IA-64 architecture. Architectures which will be analyzed include the VAX security kernel, Intel 80x86, DISCO and Alpha.

A. THE VAX SECURITY KERNEL

Developed by the Digital Equipment Corporation, the VAX security kernel is the result of a research effort that resulted in a Type I VMM for the VAX architecture [Ref. 12]. The system's hardware, software and microcode are designed to meet Trusted Computer System Evaluation Criteria (TCSEC) A1-level security requirements [Ref. 13]. The VAX security kernel runs on the VAX 8530, 8550, 8700, 8800 and 8810 processors, creates virtual VAX processors (each of which runs either the VMS or the ULTRIX-32 operating systems), and provides isolation and controlled sharing of sensitive information.

The VAX security kernel is primarily a security kernel rather than a VMM. Therefore, certain features of the VMMs, such as self-virtualization and debugging of one VM from another were not implemented in the VAX security kernel to reduce kernel complexity.

A security kernel is an implementation of the reference monitor concept and includes hardware, firmware and software [Ref. 14]. The reference monitor is an abstraction that enforces authorized subject access to objects in the system. An Object is a passive entity that contains or receives information. Access to an object means access to the information it contains. On the other hand, a subject is an active system entity such as a process that causes information to flow among objects. The reference monitor has three implementation requirements:

(1) Completeness: The reference monitor must be invoked on every access request of a subject to an object.

(2) Isolation: The reference monitor and its database must be protected from unauthorized modifications.

(3) Verifiability: The reference monitor must be well structured, small, simple and understandable, so that it can be subject to test and analysis to ensure that it performs all its functions properly.

The VAX security kernel applies mandatory and discretionary access controls to virtual machines. The kernel assigns an access class that contains a secrecy class and an integrity class to virtual machines. The secrecy class is based on the Bell and LaPadula security model [Ref. 14]. The Bell and LaPadula Model defines a policy for secrecy that restricts read and write access between a subject and an object based upon the relationship between the subject's access class and object's access class. The two properties of the Bell and LaPadula Model are:

(1) Simple Security Property (No read-up): A subject may have read access to an object only if the subject's access class is equal to or greater than the object's access class. This property ensures that secret labeled users can read secret, confidential and unclassified information, but cannot read top-secret information.

(2) * - Property (No write-down): A subject may have write access to an object only if the subject's access class is equal to or less than the object's access class. This property is required to prevent malicious software from writing down. For example, if a secret user uses a system that does not enforce the * - Property, a Trojan horse could read the secret information and write that information to unclassified files.

Policies related to the integrity or modification are often represented by the Biba security model [Ref. 14]. The Biba security model defines a policy for integrity that addresses the unauthorized modification of sensitive information by restricting read and write access between a subject and an object. To restrict read and write access between a subject and an object, the Biba Model uses integrity levels and integrity compartments. The two properties of Biba Model are:

(1) Simple Integrity Property (No write-up): A subject can have write access to an object only if the subject's integrity class is equal to or greater than the object's integrity class. This property ensures that a low integrity subject will not modify high integrity data.

(2) * - Property (No read-down): A subject can have read access to an object only if the subject's integrity class is equal to or less than the object's integrity class. This property ensures that the high integrity subject will not read low integrity data.

The VAX security kernel also supports the access control lists (ACLs) to be able to apply discretionary access controls to virtual machines. ACLs provide a list of subjects authorized to access a particular object.

Users and virtual machines are the two kinds of subjects in the VAX security kernel. A user communicates with a process called a server through a trusted path. Servers are trusted processes. The user's minimum and maximum classes, the terminal's minimum and maximum access classes, the user's discretionary access rights and privileges, and the privileges exercisable from the terminal all determine what the server can do for the user. A virtual machine runs a virtual machine operating system (VMOS), and is an untrusted subject. A user may write a program for execution inside a VM and may corrupt the VMOS without affecting the security kernel. A virtual machine could only affect the other VMs with shared disk volumes. Virtual machines can also be considered as objects because a user may want to connect to a VM. Then the security kernel establishes a connection between the user and the VM.

When a user logs into the security kernel, the VMM establishes a session between the user and the user's server. When the user wants to connect to a VM, the user uses the CONNECT command to the server and specifies the name of the VM. If the connection is authorized, the user's existing connection with the server is suspended, a new session between the user and the requested VM is established by the security kernel.

The VAX security kernel also supports objects such as real devices, disk and tape volumes and security kernel files. The real devices consist of disk drivers, tape drivers, printers, terminal lines. These devices contain or transmit information and must be controlled by Trusted Computing Base (TCB), that is, all the protection mechanisms within a computer system. Disk and tape volumes are another kind of objects. The contents of some disk volumes are controlled by a VM. Other disk volumes are VAX security kernel volumes which cannot be accessed by a VMOS. VAX security kernel volumes contain VAX security kernel files, which hold long term system databases, such as an audit log. These files are the part of the TCB, and the VM cannot reference them with the exception of the audit and error logs and crash dump files. VAX security kernel files can also be used to create virtual disk volumes.

To reduce its overall complexity, the VAX security kernel has been implemented by using the levels of abstraction approach. Each layer in the system implements some abstraction in part by making calls to lower layers. Making lower layers unaware of higher abstractions reduces the number of interactions, thus the complexity. Table 6 contains the name of each layer and brief descriptions [Ref. 11].

Highly secure systems are often hard to use because of their limited user interfaces. As a reference monitor concept requirement, the security kernel must be small and verifiable. Normally, interface features are large and hard to verify. Therefore, such features cannot be included in the security kernel. To overcome this problem, the designers created two separate command sets, secure server and secure commands. The secure server commands such as connect, disconnect, resume and show sessions are commands that control terminal connections to virtual machines. The secure commands help manage the system.

The VAX architecture was not virtualizable. Therefore, some virtualization techniques were used to make the VAX processor virtualizable. The extensions made to the VAX processor's architecture and to the processor microcode to support virtualization are described below [Ref. 12]. These techniques would be applicable to virtualizing the IA-64.

Layer Name	Definition
Users	Includes untrusted application programs that run on the VMOS and humans who communicate with the secure server through the trusted path.
VMOS	The virtual machine's operating system.
Secure Server	Implements the trusted path for the security kernel, logs users in and out, provides security-related administrative functions; everything above this layer is untrusted.
Virtual VAX	Completes the virtualization process by emulating sensitive instructions and delivering interrupts and exceptions to the virtual machine.
Kernel Interface	Implements virtual controllers for virtual I/O devices and the security function controller.
Virtual Printers	Implements virtual printers for each VM and multiplexes the real physical printers among virtual printers.
Virtual Terminals	Implements virtual terminals for each VM and manages physical terminal lines.
Volumes	Implements VAX security kernel and exchangeable volumes and provides registries of all subjects and objects.
Files-11 Files	Implements a subject of the ODS-2 file system.
Auditing	Provides the facilities for security auditing and alarms.
Higher-Level Scheduler	Creates the abstraction of level-two virtual processors (vp2s); dedicated vp2s are used by the Secure Server layer and bindable vp2s are used by virtual machines.
VM Virtual Memory	Implements the shadow page tables that are needed to support virtual memory in virtual machines.
VM Physical Memory	Manages real physical memory and assigns it to virtual machines.
I/O Services	Implements device drivers that control the real I/O devices.
Lower-Level Scheduler	Creates the abstraction of level-one virtual processors (vp1s) that are the basic unit of scheduling for the system; vp1s are intended to be very inexpensive processes for use within the kernel.
Hardware-Interrupt Handler	Immediately above the VAX hardware and modified microcode; contains the interrupt handlers for the various I/O controllers and certain CPU-specific code.

Table 5. Layers In The VAX Security Kernel Design From [Ref. 9].

1. Ring Compression

Ring compression is the most significant and security-relevant architectural change that virtualizes the protection rings [Ref. 15]. Ring compression is implemented in software. The rings of the virtual VAX processor are mapped to the rings of the real processor in such a way that ring 0 (kernel) and ring 1 (executive) of the VM are mapped to ring 1 (executive) of the real processor. Virtual user and supervisor rings are mapped to real user and supervisor rings, respectively. Ring compression was accomplished by making three extensions to the VAX architecture and modifying the sensitive instructions to make use of these extensions. There extensions include:

(1) Addition of a VM mode bit to the processor status longword (PSL<VM>)

(2) Definition of a new register called the VM processor status longword register (VMPSL)

(3) Definition of the VM-emulation exception

When the processor is executing in a virtual machine, the PSL<VM> bit is set. PSL<VM> is set only by software and is cleared only by microcode when an interrupt occurs. Software reading the PSL, whether in the virtual machine or in the virtual machine monitor, will have no opportunity to reveal that PSL<VM> is set.

Instructions either trap to the VMM security kernel for simulation of their effects or obtain their information from the VMPSL that contains the virtual ring number (rather than the real ring number).

The compression of the executive and kernel rings in the virtual machines has no security impact on the VMM since the VMM security kernel runs only in real kernel mode, and no virtual machine can access real kernel mode pages. However, ring compression could affect security within a virtual machine since the virtual machine's kernel mode is no longer protected from its executive mode.

2. Sensitive Instructions

The VAX architecture does not meet the hardware virtualization requirements. Some instructions including move processor status longword (*MOVPSL*), change-mode (*CHM*), probe (*PROBE*) and return from exception or interrupt (*REI*) are sensitive, but are not privileged instructions. Also, unprivileged instructions can read and modify the contents of page table entries (PTEs), which are sensitive data structures.

To overcome this problem and support virtualization, designers made some underlying changes to the VAX architecture as described above. Additionally, designers also changed the microcode to address the virtualization problems caused by sensitive unprivileged instructions. These changes will be discussed below [Ref. 15].

The *MOVPSL* instruction reads the processor status longword. The *MOVPSL* instruction was implemented in microcode. If PSL<VM>=0, *MOVPSL* returns the real PSL. If PSL<VM>=1, *MOVPSL* returns the VMPSL. This instruction always produces correct results and never traps.

The *CHM* instruction switches to a mode of equal or higher privilege. Four *CHM* instructions (*CHMU, CHMS, CHME, CHMK*) were defined to cause a VM-emulation trap when executed in VM mode, allowing the VMM to do the proper stack pointer and stack manipulation, examine the VM's system control block (SBC) which contains a list of addresses of interrupt handling routines, and transfer the CHM exception to the VM.

The *REI* instruction switches to a mode of equal or lower privilege. *REI* is one of the most complex instructions in the VAX architecture. Because of its complexity, REI emulation is done in software, but some useful sanity checks were left in microcode. The *REI* instruction replaces the entire PSL, therefore, the VMM must pack together the new PSL's current mode and previous mode fields and then switch to the proper VM stack.

The *PROBE* instruction is used to determine if access to a page of memory is allowed. In the modified VAX, *PROBE* reads the shadow page table entry (PTE) which contains the real physical page numbers and translated protection fields corresponding to the VM's page table entries. If the shadow PTE is valid, *PROBE* uses the protection fields and checks the requested access without trapping to the VMM. If the shadow PTE is not valid, *PROBE* traps to the VMM which updates the shadow PTE based on the VM's page table.

3. I/O Emulation

Traditional virtual machine monitors (e.g., IBM's VM/370) virtualize not only the CPU, but also the I/O hardware. Virtualized I/O hardware allows the VMOS to run unmodified. The VAX I/O hardware is difficult to virtualize because its I/O devices are programmed to read and write control and status registers (CSRs) in the I/O space of the physical memory.

To overcome this problem, the VAX security kernel I/O interface used a kernel call mechanism. In essence, the virtual machine stores I/O related parameters in its I/O space and executes a move to privileged register (MTPR) instruction to a special kernel call (KCALL) register. Then, MTPR traps to the security kernel and the I/O operation is performed. Because of the special kernel I/O interface, special untrusted virtual device drivers must be written for all VMOS.

B. INTEL 80X86 ARCHITECTURE

Intel 80x86 architecture was used to design high-assurance trusted systems such as Gemini Trusted Network Processor [Ref. 16], XTS-300 [Ref. 17], and Boeing MLS Lan [Ref. 18] during the 1980's and early 1990's.

The 80x86 architecture has many mechanisms that provide benefits for constructing secure systems. These mechanisms include: multiple operating modes (e.g., protected, virtual-8086, real); segment protection (e.g., gates); multitasking (e.g., task state segments, task switching); paging (e.g., address space management, per-page protection); input/output protection (e.g., I/O instructions), control flags and debug registers [Ref. 19].

The four hardware protection rings of the Intel architecture provide benefits for secure system construction. The protection rings solve three important problems of a general-purpose system to be used as a computer utility [Ref. 20]. First, a user can create arbitrary and protected subsystems which can be used by other users. Second, the supervisor code can be implemented in layers enforced by the hardware. Third, the user can protect himself while debugging his own programs.

The hardware components of a computer system are said to operate securely if they are used correctly. However, analysis of the Intel 80x86 processor family architectures has shown that Intel architecture has some "pitfalls" [Ref. 19]. The architectural pitfalls are not security flaws but can cause security flaws if these pitfalls are not handled by the operating system correctly. Most of the architectural pitfalls discussed are timing and storage covert channels. A covert channel transfers information from one process to another, although, transfer of the information is not expected.

The pitfalls include the TS flag, FPU context, segment accessed bit, segment attributes, page access visibility, internal register visibility, debug register values, time stamp counter, performance counters, cache and TLB timing channels and undefined values. These pitfalls were found in the early version of the 80x86 architecture (such as the 386 and 486), but most of them still remain in the current Pentium architecture.

In addition to the pitfalls listed above, the 80x86 have 102 reported flaws, 17 of which are security relevant and nine of which are denial of service flaws [Ref. 19]. System designers should consider these pitfalls when developing secure operating systems for the 80x86 architecture.

In his thesis, Robin analyzes the Intel Pentium's capability to support a secure virtual machine monitor [Ref. 11]. He concludes that even though the Intel architecture is not virtualizable, a product called FreeMWare (renamed plex86) provides something similar to a Type II VMM for the Intel architecture. However, none of the VMMs for the Intel architecture should be considered a secure VMM.

In the following paragraphs, several techniques proposed by Lawton [Ref. 21] to make the Intel architecture virtualizable, are described . These techniques are also useful for virtualization of the IA-64.

1. Pure Emulation

Pure emulation is modeling the complete x86 PC in software to run x86 operating systems and applications on non-x86 platforms. This method is used in a project called Bochs [Ref. 22]. Bochs is much more like a complete software interpreter machine rather than a VMM. Bochs emulates a large part of the CPU, related AT hardware and the BIOS, and allows DOS, Windows'95, Minix 2.0 and other x86 operating systems to be run on both x86 and non-x86 platforms. Bochs can run on Sparc, MIPS and PowerMac.

The advantage of the pure emulation technique is its portability. However, this technique has the disadvantage of significant performance degradation since no instructions are executed directly on the hardware.

2. OS/API Emulation

OS/API Emulation is the second technique proposed by Lawton. In a computer system, applications communicate with the operating system via a set of APIs. OS/API emulation is a technique that allows applications designed for a different x86 operating system to run by intercepting and emulating the behavior of the API mechanisms in the operating system. The Wine project used this technique. Wine is "an implementation of the Windows 3.x and Win32 API on top of X and Unix" [Ref. 23]. Using the OS/API emulation technique, application binaries can be run natively, that is, Windows executable files can be run in a Unix environment without any alteration. Running application binaries natively provides very good performance. But, OS/API emulation is only applicable to running x86 operating systems for which the APIs are known and have been emulated.

3. Virtualization

The third technique is called virtualization. In general, system features such as the segmentation model, protection mechanisms, and the paging unit of IA32 CPU, are designed to be used by only one operating system. Usually, the large set of instructions would cause no problems when running multiple operating systems on hardware. The virtualization technique allows the larger set of compatible instructions to execute natively. The smaller set of incompatible instructions (i.e., sensitive, unprivileged instructions) need to be detected, and their expected behavior must be emulated by the VMM.

In the virtualization technique, the instructions are analyzed until one of the following is encountered:

(1) An instruction which cannot be run natively (problem instruction);

(2) A branch instruction; or,

(3) The address of an instruction, which has been analyzed previously.

For the first two conditions, a breakpoint must be installed at the beginning of the problem instruction or branch instruction. If case three is encountered, no action is necessary since the instruction has been analyzed already.

Code is allowed to run natively, and continues to run until it encounters a breakpoint. If the reason for the breakpoint is a problem instruction, the behavior of this problem instruction is emulated by the VMM. If the reason for the breakpoint is a branch instruction, it is necessary to analyze instructions at the branch target address. Branch instructions must be dynamically controlled to ensure that execution does not branch to code that has not been analyzed. If the target address has already been analyzed, and marked as OK, then the branch instruction can be marked as OK.

The virtualization technique also addresses the possibility that some instructions which write to memory may write into the addresses of instructions which have been marked as OK. Page tables are used to write protect any page of memory in which the code has been analyzed and marked as OK.

As a summary, the Intel x86 architecture is not truly virtualizable. A product called plex86 (formerly called FreeMWare) provides something similar to a Type II VMM for the Intel x86 architecture. However it is not a good approach to implement a secure VMM as a Type II VMM. Since a Type II VMM runs as an application under a host operating system, a highly secure host operating system has to be written for the x86 architecture.

C. THE DISCO ARCHITECTURE

Another virtual machine monitor implementation is the DISCO prototype, which is also a useful guide to how the IA64 architecture virtualization might be accomplished. DISCO is an implementation of a Type I VMM, and was developed at Stanford University for the FLASH multiprocessor [Ref. 24].

The flash multiprocessor consists of a set of nodes each containing a processor, main memory, and I/O devices. The nodes are connected together with a high-performance interconnect. Although it is designed for the FLASH multiprocessor, DISCO is also available on a number of machines, such as Convex Exemplar, Silicon Graphics Origin2000, Sequent NUMAQ and Data General NUMALiine.

DISCO emulates privileged instructions, the memory management unit, and the trap architecture and allows unmodified applications and operating systems to run on the virtual machine. DISCO provides an abstraction of main memory located in a contiguous physical address space starting at address zero. For virtualization of I/O devices such as disks, network interfaces, clock and periodic interrupt timers, DISCO intercepts all communication to and from I/O devices to emulate the behavior.

Some of the key aspects of DISCO implementation which would be applicable to the virtualization of the Intel IA-64 architecture will be described below [Ref. 24].

1. Virtual CPUs

Generally, there is only one processor in a computer system. In a virtual machine monitor environment, multiple virtual machines can run on the same processor. Therefore, each VM has a virtual processor that leads the VM and VMOS to think that the virtual machine is running on the real processor.

In the DISCO project, the execution of the virtual CPU is emulated by using direct execution on the real processor. In order to schedule a virtual CPU, DISCO sets the registers of the real machine to those of the virtual CPU and jumps to the program counter of the virtual CPU. Direct execution allows the operations to run at the same speed as they would on the real processor. A challenge of using direct execution is the detection and fast emulation of those operations caused by the execution of privileged instructions.

For each virtual CPU, a data structure that functions like a process table entry in a traditional operating system is kept. This data structure contains the register values, TLB contents and other state information of a virtual CPU when it is not running on the real CPU. A VM trap causes the virtual CPU to run on the real CPU and causes the virtual machine monitor to emulate the behavior of the trap on the currently scheduled virtual processor. DISCO contains a simple scheduler that arranges the time-sharing usage of the real processor by virtual processors. The information of the currently scheduled VM is saved when the time period is up and the next virtual processor is swapped into the real processor.

2. Virtual Physical Memory

DISCO uses an additional level of address translation and maintains physical-to-machine address mappings to virtualize physical memory. Virtual machines use a physical address that starts at address zero and continues for the size of the virtual machine's memory. In the DISCO prototype, physical-to-machine translation is done by using the software reloaded TLB of the MIPS processor. Insertion of a virtual-to-physical mapping into the TLB by the operating system requires emulation of this insertion operation by translating the physical address into the associated machine address and inserting this machine address into the TLB.

3. Virtual I/O Devices

To virtualize I/O devices, the VMM must intercept all device accesses from virtual machines and forward these device accesses to physical devices. The DISCO designers have found that it is easier to use one universal device driver for each type of device rather than to use a real device driver of every device. Each device defines a "monitor call" which is used to pass all command arguments to the VMM in a single trap. Devices such as disks and network interfaces require a direct memory access (DMA) map as their arguments. The VMM must intercept such DMA requests and translate a physical address into a machine address. Then device drivers interact directly with the physical device.

4. Virtual Network Interface

Virtual machines use standard distributed protocols such as network file system (NFS) to communicate with each other. DISCO has designed a virtual subnet that allows communication between the virtual machines. The virtual subnet and interfaces use copy-on-write mappings to transfer data between virtual machines in order to reduce copying overhead and to allow for memory sharing.

D. ALPHA ARCHITECTURE

The Alpha is a 64-bit load and store RISC machine [Ref. 25]. The Alpha is designed with particular emphasis on clock speed, multiple processors and multiple instructions issue, all of which have an effect on processor performance. Similar to IA-64 architecture, the Alpha architecture was designed as a 64-bit architecture; all operations are performed between 64-bit registers. The instructions are very simple and are 32 bits in length. Memory operations are either loads or stores. The Alpha supports the Open VMS Alpha, DIGITAL UNIX and Windows NT Alpha.

One of the goals of the Alpha architecture is to implement functions without microcode. However, it is desirable to provide an architectural interface to these functions that will be consistent across the entire processor family. The Privileged Architecture Library (PALcode) provides this interface. The implementation of the PALcode differs for each operating system. PALcode implements some low-level hardware support functions that are too complex, too costly, or otherwise impractical to implement directly in the processor's hardware and which cannot be achieved by a normal operating system. These low-level hardware support functions include:

(1) Privileged instructions

(2) Atomic operations such as context switching, returns from exceptions or interrupts that require complete access to the underlying hardware

(3) Complex sequences such as translation buffer fill routines

(4) Instruction emulation without hardware support

(5) Power-up initialization and booting

In some architectures, microcode handles these low-level hardware support functions. The Alpha architecture is designed to not to mandate the use of microcode for reasonable implementations. Therefore, PALcode provides a special environment between the processor and the operating system to handle these hardware functions. Figure 14 shows the role of PALcode in Alpha architecture.

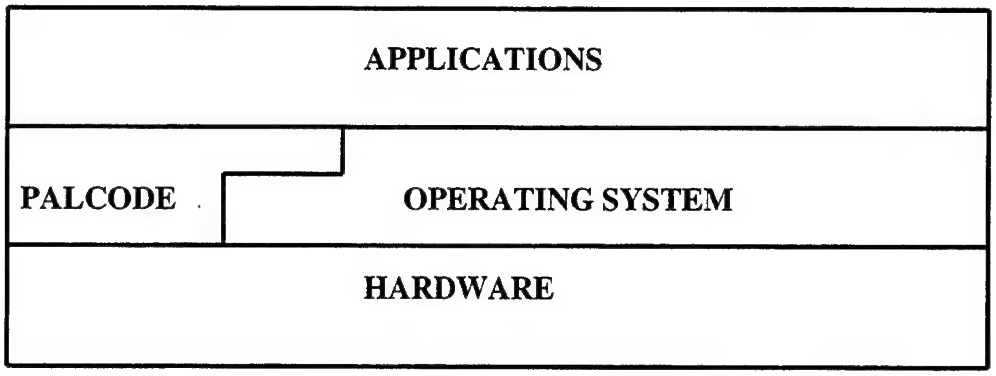

Figure 14. PALcode Role in Alpha Architecture From Ref. [25].

Applications have access to PALcode and to the operating system. This capability allows the applications to have direct access to low-level hardware support functions through PALcode, or to interact with them through the operating system. The operating system can either have direct access to the low-level hardware support functions or can give control to PALcode.

PALcode runs in a special privileged environment called PALmode. PALcode is written with the standard Alpha instruction set plus five reserved opcodes to implement

PALcode-specific instructions. These opcodes have implementation-specific extentions that provide low-level hardware support functions for changing states, reading and modifying hardware control registers. PALcode-specific instructions:

(1) Perform physical memory load or store operations without invoking memory management routines

(2) Move data to and from internal processor registers

(3) Switch the PALcode environment to the native-mode environment

PALcode is invoked at specific entry points and certain well-defined conditions. PALcode can be invoked by the following hardware and software operations:

(1) Reset

(2) Hardware exceptions (e.g., arithmetic trap)

(3) Memory management exceptions

(4) Interrupts

(5) CALL-PAL instructions

The Alpha architecture allows two methods of entry into the PALmode environment:

(1) Hardware-detected: PALcode responds to a hardware event (e.g., reset, interrupts, arithmetic exceptions).

(2) Software-initiated: PALcode responds to a privileged or unprivileged CALL-PAL function.

The Alpha architecture is designed to support virtualization [Ref. 26]. PALcode provides an interface without microcode for privileged instructions. All privileged and sensitive instructions are required to be implemented by PALcode. PALcode is required to be replaceable, allowing per-operating system code to achieve high performance. The Alpha architecture is designed such that the operating system can ensure that there is no way to bypass the protection mechanisms. Even "UNPREDICTABLE" results or occurrences (i.e., the results or occurrences that may vary from moment to moment, implementation to implementation, and instruction to instruction within implementations) are constrained, therefore, security and virtualization are supported. Additionally, as a result of "UNDEFINED" operations (i.e., operations that may vary from moment to moment, implementation to implementation, and instruction to instruction within implementations), the processor may contain wrong information or may lose information. However, these UNDEFINED operations will not cause security problems since they can only be executed by privileged software [Ref. 26].

E. CONCLUSION

After analyzing current VMM architectures and virtualization techniques, it is clear that some slight modifications to the Intel IA-64 architecture could make virtualization easier. First, all 18 sensitive, unprivileged instructions could be redesignated as privileged instructions. However, this solution may cause performance degradation problems, because unprivileged instructions will now trap to the VMM, whereas they did not previously.

The second modification could be to implement a trap on op-code instruction [Ref. 7]. By adding a new instruction to the architecture, an operating system could treat the sensitive unprivileged instructions as if they were privileged. This modification makes virtualization easier and does not affect the current operating systems.

Without these two modifications, virtualization will be much harder. Additional code is required in order to force the sensitive, unprivileged instructions to trap to the VMM. Additional code may cause security problems, because it would violate the verifiability requirements of the reference monitor concept since the security kernel would no longer be simple, small and understandable.

Additionally, without modification, a processor must check every instruction before the instruction executes to determine if the instruction is one of the eighteen problem instructions. (Of course, undocumented instructions exist, the VMM will not operate correctly, because the VMM will not recognize the undocumented instruction.)

THIS PAGE INTENTIONALLY LEFT BLANK

VI. CONCLUSION

This thesis analyzed the Intel IA-64 architecture's capability to support a secure virtual machine monitor. The thesis began by studying the Intel IA-64 architecture. Innovations and new features as well as the application and system state organization and addressing, protection mechanisms of the IA-64 architecture were studied. Then, virtual machine and virtual machine monitor concepts were introduced. Three types of the virtual machine monitors and the hardware requirements for each type were covered. After that, analysis of the Intel IA-64 architecture's support for a virtual machine monitor was done. The IA-64 architecture was mapped to the hardware requirements. A large part of this analysis involved looking at each of the documented instructions of the IA-64 architecture. Approximately 150 instructions were analyzed to determine if the IA-64 architecture is virtualizable. The analysis showed that although the IA-64 architecture meets three main hardware requirements to support a Type I VMM, it contains sensitive unprivileged instructions. Eighteen sensitive unprivileged instructions violate the hardware virtualization requirements, preventing the IA-64 architecture from being virtualizable.

Although the IA-64 architecture is not virtualizable, some other architectures were examined to determine what changes could help the IA-64 architecture be virtualizable. Several virtualization techniques that might be applicable to virtualizing the IA-64 architecture were discussed.

A secure VMM on the IA-64 architecture would offer several benefits. First, it could eliminate the need for dedicated systems for each security level. Second, PC users would be able to work in a high assurance system and still run COTS operating systems.

The Intel IA-64 architecture has features that might be used to construct secure systems. However, before trying to implement a secure Type I VMM for the Intel IA-64 architecture, some slight modifications to the processor are needed.

A. FUTURE WORK

Two areas of future work seem feasible. First, IA-64 designers might look at the VAX security kernel architecture in more detail and determine how this architecture was modified to support virtual machines. Exploration of the feasibility of implementing these modifications on the Intel IA-64 architecture would be useful.

Second, it would be useful to look at the source code of the plex86 product which provides a Type II VMM for the Intel x86 architecture. Analysis of how this source code provides virtualization would be a good opening for developing a VMM for the IA-64 architecture.

LIST OF REFERENCES

1. Intel Corporation, *Intel IA-64 Architecture Software Developer's Manual, Volume 1: IA-64 Application Architecture,* July 2000.

2. Intel Corporation, *Intel IA-64 Architecture Software Developer's Manual, Volume 2: IA-64 System Architecture,* July 2000.

3. Intel Corporation, *Intel IA-64 Architecture Software Developer's Manual, Volume 3: Instruction Set Reference,* July 2000.

4. Intel Corporation, *Intel IA-64 Architecture Software Developer's Manual, Volume 4: Itanium^{TM} Processor Programmer's Guide,* July 2000.

5. Geva, Robert, and Morris, Dale "IA-64 Architecture Disclosures White Paper" [http://www.cs.nmsu.edu/~rvinyard/itanium/docs/ia64_arch_wp.pdf].

6. Diefendorff, Keith, "HP, Intel Complete IA-64 Rollout- Virtual Memory, Interrupts More Conventional Than ISA," *Microprocessor Report*, April 2000.

7. Goldberg, Robert, *Architectural Principle for Virtual Computer Systems,* Ph.D. Dissertation, Harvard University, Cambridge, Massachusetts, October 1972.

8. Popek, Gerald J. "Formal Requirements for Virtualizable Third Generation Architectures," *Communications of the ACM, v*. 17.7, pp. 412-421, July 1974.

9. Sun Microsystems, Security in Java, [http://www.asert.com/JavaTut.html]. 1999.

10. Transmeta Corporation, The Smart Microprocessor for Mobile Internet Computing, [http://www.transmeta.com/crusoe]. 2000

11. Robin, Scott, *Analyzing the Intel Pentium's Capability to Support a Secure Virtual Machine Monitor*, Master's Thesis, Naval Postgraduate School, Monterey, California, September 1999.

12. Karger, Paul A., Zurko, Mary Ellen, Bonin, Douglas W., Mason Andrew H., Kahn, Clifford E., "A VMM Security Kernel for the VAX Architecture," In *Proceedings of the 1990 IEEE Security and Privacy Conference*, 1990.

13. Department of Defense, "*Department of Defense Trusted Computer System Evaluation Criteria," DOD 5200.28-STD*, December 1985.

14. Course Notes for CS3600 Introduction to Computer Security, Naval Postgraduate School Center for Information Systems Security Studies and Research, 1999.

15. Hall, Judith, and Robinson, Paul T., "Virtualizing the VAX Architecture," *Proceedings of the 18th International Symposium on Computer Architecture,* pp.380-389, Toronto, Canada, May 1991.

16. National Security Agency Report NCSC-FER-94/34, *Final Evaluation Report: Gemini Computer, Inc Gemini Trusted Network Processor Version 1.01,*28 June 1995.

17. National Security Agency Report CSC-EPL-92/003, *Final Evaluation Report: HFS Incorporated, XTS-200,* 27 May 1992.

18. National Security Agency Report CSC-EPL-91/005, *Final Evaluation Report: Boeing Space and Defense Group, MLS LAN Secure Network Server System,* 28 August 1991.

19. Sibert, Olin, Porras, Phillip A., Lindell, Robert, "The Intel 80x86 Processor Architecture: Pitfalls for Secure Systems," In *Proceedings1995 IEEE Symposium on Security and Privacy,* Oakland, CA, May 1995.

20. Schroeder, Michael D. and Saltzer, Jerome H., "A Hardware Architecture for Implementing Protection Rings," *Communications of the ACM,* v. 15.3, pp.157-169, March 1972.

21. Lawton, Kevin, "Running Multiple Operating Systems Concurrently on an IA32 PC Using Virtualization Techniques." [http://www.freemware.org]. June 1999.

22. Lawton, Kevin, "Welcome to the 'Bochs Software Company' Home Page." [http://bochs.com] July 1999.

23. Wine, "Wine Development HQ." [http://winehq.com]. July 1999.

24. Bugnion, Edouard, Devine, Scott, Govil, Kinshuk, and Rosenblum, Mendel, "Disco: Running Commodity Operating Systems on Scalable Multiprocessors," *ACM Transactions on Computer Systems, v.* 15.4, pp. 412-447, November 1997.

25. Alpha Architecture Handbook, Technical Report Order Number: ECQD2KC-TE, October 1998.

26. Robin, Scott J., Irvine, Cynthia E., "Analysis of the Intel Pentium's Capability to Support a Secure Virtual Machine Monitor," [http://www.cs.nps.navy.mil/people/faculty/irvine/publications/2000/VMM-usenix00-0611.pdf]. 1999.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
   8725 John J. Kingman Road, Suite 0944
   Ft. Belvoir, VA 22060-6218

2. Dudley Knox Library 2
   Naval Postgraduate School
   411 Dyer Road
   Monterey, CA 93943-5101

3. Chairman, Code EC 1
   Electrical and Computer Engineering Department
   Naval Postgraduate School
   Monterey, CA 93943-5100

4. Chairman, Code CS 1
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943-5000

5. Prof. Cynthia Irvine 3
   Computer Science Department
   Naval Postgraduate School
   Monterey, CA 93943

6. Senior Lecturer Frederick W. Terman 1
   Electrical and Computer Engineering Department
   Naval Postgraduate School
   Monterey, CA 93943

7. Deniz Kuvvetleri Komutanligi 2
   Personel Daire Baskanligi
   Bakanliklar
   Ankara, TURKEY

8. Deniz Kuvvetleri Komutanligi 1
   Kutuphane
   Bakanliklar
   Ankara, TURKEY

9. Deniz Harp Okulu Komutanligi 1
Kutuphane
Tuzla
Istanbul, TURKEY

10. Carl Siel 1
Space and Naval Warfare Systems Command
PMW 161
Building OT-1, Room 1024
4301 Pacific Highway
San Diego, CA 92110-3127

11. Commander, Naval Security Group Command 1
Naval Security Group Headquarters
9800 Savage Road
Suite 6585
Fort Meade, MD 20755-6586

12. Ms. Deborah M. Cooper 1
Deborah M. Cooper Company
P.O. Box 17753
Arlington, VA 22216

13. Ms. Louise Davidson 1
N643
Presidential Tower 1
2511 South Jefferson Davis Highway
Arlington, VA 22202

14. Mr. William Dawson 1
Community CIO Office
Washington DC 20505

15. Capt. James Newman 1
N64
Presidential Tower 1
2511 South Jefferson Davis Highway
Arlington, VA 22202

16. Mr. Richard Hale 1
Defense Information Systems
Agency, Suite 400
5600 Columbia Pike
Falls Church, VA 22041-3230

17. Barbara Flemming 1
Defense Information Systems
Agency, Suite 400
5600 Columbia Pike
Falls Church, VA 22041-3230

18. CDR Keith Fuller 1
Defense Information Systems
Agency, Suite 400
5600 Columbia Pike
Falls Church, VA 22041-3230

19. Kadir Karadeniz 2
Inkilap Mahallesi Duz Sokak No:23/3
Umraniye- Istanbul, TURKEY